VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

NÁSTROJE PRO TVORBU MULTIPLATFORMNÍCH APLIKACÍ
TOOLS FOR CREATING CROSS-PLATFORM APPLICATIONS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Martin Smíšek**
AUTHOR

VEDOUCÍ PRÁCE — **doc. Ing. Ivo Lattenberg, Ph.D.**
SUPERVISOR

BRNO 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**
Ústav telekomunikací

Student: Bc. Martin Smíšek ***ID:*** 153163
Ročník: 2 ***Akademický rok:*** 2016/17

NÁZEV TÉMATU:

Nástroje pro tvorbu multiplatformních aplikací

POKYNY PRO VYPRACOVÁNÍ:

Vytvořte multiplatformní aplikaci s využitím technologie Xamarin Forms. Cílovou platformou by měl být alespoň Android, Windows 10 a Windows Phone. Aplikace by měla sloužit jako komfortní a uživatelsky přívětivá dálková správa kotle pro vytápění domácnosti. Aplikace bude v lokální síti komunikovat přes převodník PRE10 s řídicí jednotkou PT41. V případě desktopové aplikace je možná i komunikace přes USB rozhraní. K jednotce PT41 budou připojeny teplotní senzory z jednotlivých místností, kotel a oběhové čerpadlo. V aplikaci uvažujte možné zřetězení více řídicích jednotek pro správu více místností. Aplikace bude zobrazovat teplotu ve sledovaných místnostech a bude umožňovat konfigurovat programy pro ovládání kotle na základě teploty v jednotlivých místnostech.

DOPORUČENÁ LITERATURA:

[1] WATSON, B., C# 4.0 - řešení praktických programátorských úloh, Zoner Press, 2010, 656 s., ISBN 978-8--7413-094-6

[2] VIRIUS, M., C# 2010 Hotová řešení, Computer Press, 2012, 424 s., ISBN 978-80-251-3730-7

Termín zadání: 1.2.2017 ***Termín odevzdání:*** 24.5.2017

Vedoucí práce: doc. Ing. Ivo Lattenberg, Ph.D.
Konzultant:

doc. Ing. Jiří Mišurec, CSc.
předseda oborové rady

Abstrakt

Cílem práce je rozbor prostředí Xamarin.Forms a implementace aplikace na něm založené. Pro vývoj bylo použito vývojového prostředí Visual Studio. Byly využity externí knihovny Skia.Sharp, Prism, Acr.UserDialogs, sqlite-net-pcl a rda.SocketsForPCL. V implementované aplikaci je více než 90 % kódu sdíleno mezi platformami. Je také zajištěna podpora vícejazyčnosti a konzistence zadaných dat. Aplikace je funkční na zařízeních s opračním systémem Android a desktpových i mobilních zařízeních s Windows 10. Hlavním přínosem této práce je objasnění principů Xamarin.Forms s jejich demonstrací ve formě vytvořené aplikace.

Klíčová slova

Acr.UserDialogs, Android, Multiplatformní aplikace, Prism, PT41, rda.SocketsForPCL, Skia.Sharp, sqlite-net-pcl, Visual Studio, Windows 10, Xamarin.Forms

Abstract

The goal of the thesis is to analyse the Xamarin.Forms framework and implement an application based on it. Visual Studio was used as an development environment. External libraries Skia.Sharp, Prism, Acr.UserDialogs, sqlite-net-pcl and rda.SocketsForPCL were used. More than 90% of the application code is shared among all platforms. The application is multilingual and able to keep persistent data. Supported operation systems are Android and Windows 10 covering mobile and desktop devices. The main benefit of the thesis is clarification of the Xamarin.Forms framework principles demonstrated by developed application.

Keywords:

Acr.UserDialogs, Android, Multiplatform applications, Prism, PT41, rda.SocketsForPCL, Skia.Sharp, sqlite-net-pcl, Visual Studio, Windows 10, Xamarin.Forms

SMÍŠEK, M. *Nástroje pro tvorbu multiplatformních aplikací.* Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2017. 117 s. Vedoucí diplomové práce doc. Ing. Ivo Lattenberg, Ph.D..

Prohlášení

Prohlašuji, že svou diplomovou práci na téma Nástroje pro tvorbu multiplatformních aplikací jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následku porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonu (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

V Brně dne ....

Martin Smíšek

Poděkování

Děkuji vedoucímu diplomové práce doc. Ing. Ivo Lattenbergovi, Ph.D. za pomoc a ochotu při řešení problémů spojených s diplomovou prací.

V Brně dne ....

Martin Smíšek

SIX research centre
sensor, information and communication systems

Faculty of Electrical Engineering
and Communication

Brno University of Technology
Technicka 12, CZ-61600 Brno, Czechia

http://www.six.feec.vutbr.cz

Výzkum popsaný v této diplomové práci byl realizovaný v laboratořích podpořených projektem Centrum senzorických, informačních a komunikačních systémů (SIX); registrační číslo CZ.1.05/2.1.00/03.0072, operačního programu Výzkum a vývoj pro inovace.

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

2007–13
OP Výzkum a vývoj pro inovace

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Diplomová práce se zabývá rozborem vývoje multiplatformních aplikací. Multiplatformní aplikace jsou takové, jejichž část kódu je možné sdílet mezi platformami. Výhodou vývoje takových aplikací je časová úspora vzniklá odstraněním nutnosti implementace separátních projektů pro jednotlivé platformy. Další výhodou je snížené množství znalostí a vývojových prostředí potřebných k vývoji. Práce je zaměřena na technologii Xamarin, již detailně popisuje.

Práce sestává z jedenácti kapitol. V první části práce zahrnující osm kapitol je rozebráno prostředí Xamarin.Forms. V první kapitole je pojednáno o historii multiplatformních aplikací. Kapitola je zaměřena zejména na historii vývoje aplikací pro operační systémy Windows, Android a iOS. Ve druhé kapitole je provedena analýza způsobů vytváření projektů v Xamarin.Forms. Zároveň je pojednáno o struktuře vytvořeného projektu a jsou rozebrány soubory v něm obsažené. Třetí kapitola se zabývá rozborem první části multiplatformních visuálních elementů. Popisuje třídy, které definují elementy a způsob práce s nimi. V druhé části je představen jazyk XAML (*Extensible Application Markup Language*) vhodný pro tvorbu uživatelského rozhraní a jsou uvedeny jeho funkce a principy. Data binding a návrhový vzor MVVM (*Model View View-Model*) jsou obsahem čtvrté kapitoly. Jedná se o přístupy umožňující zpřehlednění kódu v mnoha ohledech, které jsou popsány. Pátá kapitola je zaměřena na visuální elementy sloužící k umístění jiných visuálních elementů dle jistých pravidel. Ostatní visuální elementy jsou popsány v šesté kapitole, která rozebírá všechny typy stránek a možnosti navigace mezi nimi. Sedmá kapitola obsahuje způsoby přístupu k platformě specifickému kódu v prostředí Xamarin. V této kapitole je pojednáno i o vývoji vlastních uživatelských elementů, kdy se platformě specifického kódu hojně využívá. Transformace a animace umožňující provedení různých statických či dynamických visuálních efektů jsou náplní osmé kapitoly. Tato kapitola také objasňuje použití tříd TriggerBase, Behaviour a jejich potomků. Tyto třídy rozšiřují funkcionalitu jazyka XAML, čímž redukují potřebný kód jazyka C#. Vybrané kapitoly jsou doprovázeny kódem, sloužícím pro demonstraci popsaných principů.

Další tři kapitoly (devátá až jedenáctá) se zabývají návrhem a popisem implementace aplikace PT41. Tímto způsobem je presentováno využití teoretických poznatků první části. Aplikace PT41 slouží k ovládání jednotek PT41 firmy Elektrobock. V desáté kapitole je proveden rozbor jednotek PT41. Je uvedena jejich funkcionalita i možnosti jejich připojení k externím zařízením. Jedenáctá kapitola pojednává o využitých knihovnách dostupných v podobě NuGet balíčků. Nejdříve je popsána knihovna prism využívána pro podporu návrhového vzoru MVVM. Následuje popis knihoven pro zobrazování platformě specifických dialogů, síťovou komunikaci a práci s databází. Popsána je také bohatá grafická knihovna

Skia.Sharp. Poslední kapitola se zabývá rozborem implementovaných projektů a tříd, které společně tvoří aplikaci PT41. První část kapitoly je věnována vytvořeným uživatelským elementům s využitím knihovny Skia.Sharp i pomocí principů uvedených v teoretické části. Dále jsou rozebrány přístupy k práci s databází, implementace návrhového vzoru MVVM a jsou popsány a zobrazeny jednotlivé stránky tvořící aplikaci. Na závěr kapitoly je pojednáno o podpoře vícejazyčnosti aplikace za použití externího nástroje a možnostech ladění a testování aplikací pro Xamarin.Forms. Jsou také rozebrány možnosti nasazení na fyzické i virtuální zařízení.

Přestože prostředí Xamarin umožňuje vývoj pro platformy Windows, Android a iOS, jsou doprovodné obrázky zobrazeny pouze na zařízeních Windows a Android, kdy zařízení Windows je zobrazeno na levé straně obrázku (není-li uvedeno jinak). Veškerý kód byl implementován za použití vývojového prostředí Visual Studio.

1 Historie multiplatformních aplikací

Svět IT (*Information Technology*) se velice rychle rozvíjí. Velký rozvoj je patrný v odvětví mobilních zařízení, doprovázený vznikem tabletů, chytrých telefonů, chytrých hodinek a dalších. Na tyto změny bylo třeba reagovat i změnou způsobu programování. K těmto změnám došlo zejména proto, že aplikace pro mobilní zařízení musí být schopny komunikovat s uživatelem pomocí doteků obrazovky. Hlavními hráči na trhu s mobilními zařízeními jsou firmy Apple se zařízeními iPhone a iPad využívajícími operačním systémem iOS a Google s širokým spektrem zařízení využívajících operačním systémem Android založeném na linuxovém jádře. I přes jejich převahu na trhu je zde i třetí hráč. Jedná se o firmu s dlouholetou tradicí ve výpočetní technice, firmu Microsoft. Microsoft započal vývoj mobilních zařízení v roce 2010 s chytrými telefony Windows 7. Desktopové a mobilní aplikace ovšem neměly společné API (*Application Programming Interface*) ani operační systém. S příchodem Windows 8 bylo vytvořeno jádro společné pro mobilní a desktopovou verzi operačního systému. Verze 8.1 poprvé nabídla společné API pro tyto obě verze. Konečně verze 10 uvedla UWP (*Universal Windows Platform*). UWP aplikace pracují nejen na mobilních a desktopových zařízeních ale také na zařízeních určených pro podporu IoT (*Internet of Things*), holografických zařízeních, Xboxech a všech dalších s operačním systémem 10, který je na všech těchto zařízeních stejný.

Další snahou bylo spojit vývoj pro všechny platformy stejně jako to UWP umožnila pro všechna zařízení s operačním systémem Windows 10. Takovýto přístup ovšem není jednoduchý. Uživatelská rozhraní jsou pro jednotlivé platformy různá s různým API, byť založena na stejných principech. Aplikace jsou také psány v jiných programovacích jazycích (Objective-C pro iPhone a iPad, Java pro Android a C# pro Windows). Z těchto jazyků se jako nejvhodnější jeví C#, jelikož je to jazyk objektově orientovaný, podporuje události, lambda funkce, LINQ dotazy, má dobrou podporu asynchronních operací a další. Tento jazyk byl ovšem původně cílen pouze na zařízení s OS (*Operation System*) Windows. Firma Ximian ale začala vyvíjet projekt Mono, jehož účelem bylo překladač jazyka C# a .NET spustitelný na OS Linux. Tato společnost byla koupena společností Novel a v roce 2011 vznikl projekt Xamarin za účelem rozšíření myšlenky Mono na mobilní zařízení. Tato snaha vyústila v překladač jazyka C# zvaný Roslyn v roce 2014. O dva roky později byl Xamarin koupen firmou Microsoft a v současné době jej lze zdarma využívat s integrací do vývojového prostředí Visual Studia či Xamarin Studia.

Během vývoje Xamarinu vznikly tři .NET knihovny. Xamarin.Mac, Xamarin.iOS a Xamarin.Android. Tyto knihovny obsahují .NET verze API pro Mac, iOS a Andoid. V jazyce C# je tedy možné psát aplikace pro všechny tyto platformy plus platformy s OS Windows. Vývoj je možné provádět ve Visual Studiu (na zařízeních s OS Windows) nebo

Xamarin Studiu (na zařízeních s iOS). V případě vývoje pro iPhone a iPad za použití Visual Studia je ovšem nutné mít připojený MAC (*Apple Macintosh Computer*). Xamarin Studio nepodporuje vývoj aplikací pro platformu Windows.

Vývoj aplikací ukázal, že dobrým postupem při vývoji aplikací je separace uživatelského rozhraní, logiky a dat. Vznikl tak návrhový vzor MVC (*Model-view-controller*), později modifikovaný na model MVVM, používaný v moderních aplikacích firmy Microsoft. Vzor MVVM dělí aplikace na model, sestávající z datových tříd, view pro uživatelské rozhraní a view-model, který řídí přenos dat mezi těmito dvěma komponentami.

V multiplatformním světě se odděluje i kód nezávislý na platformě a kód závislý na platformě. Kód nezávislý na platformě je vložen do separátního projektu. Separátní projekt může být typu SAP (*Shared Asset Project*) nebo PCL (*Portable Class Library*). Aplikace pro Android, iOS i Windows mohou k tomuto sdílenému projektu přistupovat. Android a iOS aplikace navíc přistupují k `Xamarin.Android` nebo `Xamarin.iOS` knihovnám zmíněným výše, které jim zpřístupní platformě specifické API. Xamarin ovšem pokročil ještě dále a umožnil sdílet i kód uživatelského rozhraní pomocí Xamarin.Forms, který je hlavním předmětem této práce.

Xamarin.Forms však není jedinou prostředím pro vývoj multiplatformních aplikací. Tyto prostředí by se daly rozdělit do dvou skupin. První z nich (ke kterým patří i Xamarin.Forms) slouží pro vývoj aplikací pomocí programovacích jazyků, které byly historicky využívány pro vývoj desktopových či mobilních aplikací (C#, C++, Pyton, ObjectiveC, Java a další). Druhá skupina využívá programovací jazyky využívané pro webové aplikace (JavaScript, HTML5 (*HyperText Markup Language*), CSS (*Cascading Style Sheets*) či AngularJS). Do první skupiny patří mimo Xamarin.Forms například Corona a Qt. Zástupci druhé skupiny jsou Sencha, PhoneGap či Appcelerator Titanium.

Corona je optimalizována pro vývoj her pomocí jazyka Lua napsaného v jazyce C. Podpora platforem je velká (Android, iOS, OS X, Windows, AppleTV a Kindle). Základní verze SDK (*Software Development Kit*) je zdarma. Qt umožňuje vývoj v jazyce C++. Jako IDE (*Integrated Development Environment*) je použit Qt Creator .Qt Designer slouží pro tvorbu uživatelského rozhraní. Qt Linguist pro podporu vícejazyčných aplikací. Dokonce byl vyvinut i add-in do Visual Studia. Měsíční cena Qt se pohybuje kolem 300 dolarů na vývojáře.

Hlavní produkt firmy Sencha Ext JS 5 slouží pro vývoj aplikací v jazyce HTML5. Nabízí také nástroje jako Sencha Architect pro vývoj uživatelského rozhraní, Sencha Inspector pro debuggování a další. Kód jazyka HTML5 je pak třeba přeložit například pomocí nástroje PhoneGap. Do týmu pěti vývojářů je cena licence Sencha 3225 dolarů za rok. PhoneGap je vyvíjen firmou Adobe a umožňuje překlad kódu jazyka HTML, CSS a JavaScript

a následné nasazení na zařízení platforem Android, iOS, WindowsPhone a webOS. Jeho výhodou je, že je zdarma.

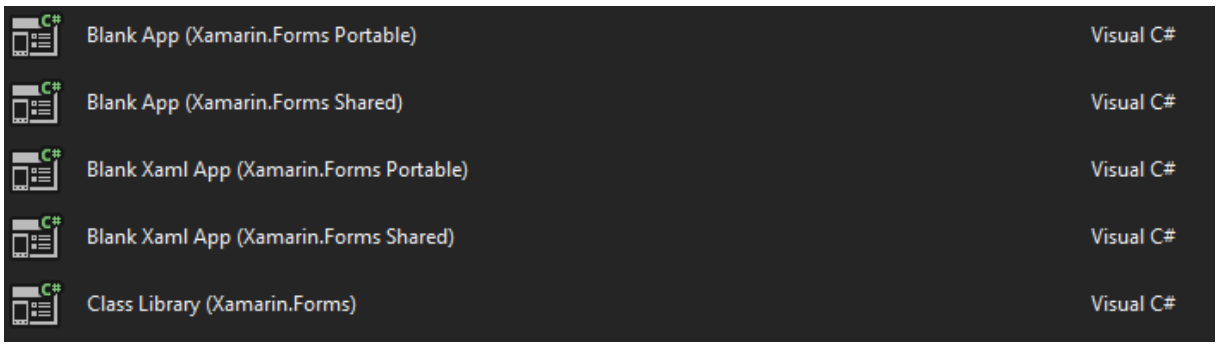
Appcelerator Titanium umožňuje vývoj v jazyce JavaScript. I tato platforma má vlastní vývojové prostředí Appcelerator Studio, prostředí pro návrh uživatelského rozhraní (App Designer), Hyperloop umožňující přístup k iOS či Android API za použití jazyka JavaScript a další nástroje. Cena se liší v závislosti na verzi od 36 dolarů za měsíc [22][1]. Jak vyplývá z textu, mnoho multiplatformních aplikací je placených. To hraje ve prospěch Xamarin.Forms. Jeho výhodou je i fakt, že za ním stojí firma Microsoft a jazyky C# a XAML, se kterými je mnoho programátorů zvyklých pracovat a jsou osvědčené. Na druhou stranu vývojáři webových aplikací budou mít blíže k jeho alternativám.

2 Výchozí aplikace Xamarin.Forms

Jak již bylo uvedeno, multiplatformní aplikace Xamarin.Forms budou vytvářeny pomocí Visual Studia 2015. Následující kapitola popisuje vytvoření nového projektu a popis souborů v něm obsažených. V kapitole bylo čerpáno z [13][24].

2.1 Šablony projektu

Při vytváření nového Xamarin.Forms projektu je nabízeno několik šablon viz Obr. 1.



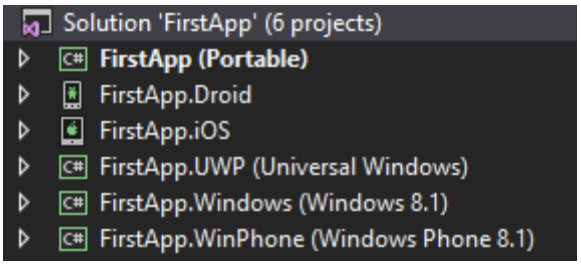
Obr. 1 Možné typy projektů ve Visual Studiu

Poslední možnost, Class Library slouží pro vytvoření knihovny pro Xamarin.Forms. Po zvolení Portable je vytvořen sdílený projekt typu PCL (*Portable Class Library*). Vznikne tedy knihovna DLL (Dynamic-Link Library), na kterou se budou platformě specifické projekty odkazovat. Po vytvoření projektu typu SAP (*Shared Asset Project*) je vytvořen společný projekt, který se při překladu stává součástí každého z platformě specifických projektů. Platformě specifický kód může být v SAP oddělen například direktivami preprocesoru.

Verze se slovem Xaml se liší pouze v tom, že obsahují ve sdíleném projektu soubor nazvaný MainPage.xaml a MainPage.xaml.cs definující společné uživatelské rozhraní pro všechny platformy.

2.2 Rozbor vytvořené aplikace

V případě volby PCL i SAP je po potvrzení vytvořen sdílený projekt a dalších pět projektů pro jednotlivé platformy (iOS, Android, UWP, Windows 8.1 a Windows Phone 8.1). Poslední tři lze souhrnně nazvat Windows projekty. Strukturu aplikace nazvané FirstApp založené na šabloně Blank Xaml App (Xamarin.Forms.Portable) lze vidět na Obr. 2.



Obr. 2 Struktura projektů nově vytvořené aplikace

Během vytváření projektu jsou staženy NuGet balíčky Xamarin.Forms knihoven a jsou vytvořeny reference všech projektů na tyto balíčky. Tyto balíčky nemusí být při vytvoření projektu aktualizované a je proto vhodné je ihned po vytvoření projektu aktualizovat. Dále je třeba nastavit konfiguraci projektu v okně Build – Configuration Manager. Je třeba nastavit možnost Build pro všechny projekty a pro platformě specifické projekty i možnost Deploy. Dále je třeba zvolit verzi systému (ARM, x64, x86, případně Any CPU pro podporu všech) a pro iOS, zda bude aplikace spuštěna v emulátoru nebo na fyzickém zařízení.

Pro výběr spouštěné platformě specifické aplikace, ji stačí nastavit jako výchozí projekt. Pokud není třeba vyvíjet pro všechny platformy, lze příslušný projekt odstranit.

Pro podporu definice GUI (*Graphical User Interface*) vzniklo Xamarin.Forms API obsahující z knihovny `Xamarin.Forms.Core` a `Xamarin.Forms.Xaml`. Na základě platformy volá `Xamarin.Forms.Core` platformě specifickou knihovnu `Xamarin.Form.Platform`. Tyto knihovny jsou implementovány jako takzvané renderery, které převádí elementy uživatelského rozhraní definované v Xamarin.Forms na elementy platformě specifické. Například element `Xamarin.Forms.Label` bude převeden na `UILabel` v prostředí iOS, `TextView` v prostředí Android a `TextBlock` v prostředí Windows. Tímto způsobem je zajištěno, že elementy uživatelského rozhraní budou vypadat na každé platformě jím specifickým způsobem. Je také možné vytvářet vlastní renderery a tím tedy i vlastní elementy uživatelského rozhraní specifické pro jednotlivé platformy.

Po spuštění výchozího projektu na libovolné z platforem se ve středu obrazovky zobrazí text „Welcome to Xamarin Forms“. Ve sdílené knihovně je třída `App`, která dědí z třídy `Application` a třída `MainPage`, která dědí z `ContentPage`.

Třídy `MainPage` a `App` jsou označeny modifikátorem `partial`. Pro každou z těchto tříd jsou vytvořeny dva soubory MainPage.xaml respektive App.xaml pro kód jazyka XAML a MainPage.xaml.cs a App.xaml.cs pro kód na pozadí.

Třída `App` v App.xaml.cs přiřazuje vlastnosti `MainPage` instanci třídy `MainPage`. Kód na pozadí obsahuje také metody, které se volají při změně stavu aplikace (někdy zvané životní cyklus aplikace).

```
    public partial class App : Application
   {
        public App()
        {
            InitializeComponent();
            MainPage = new FirstApp.MainPage();
        }
        protected override void OnStart() {}
        protected override void OnSleep() {}
        protected override void OnResume() {}
    }
```

Metoda `OnStart` je vyvolána po ukončení konstruktoru aplikace. Jedná se o vhodné místo pro načtení uložených dat a nastavení. Metoda `OnSleep` je vyvolána z důvodu neaktivity uživatele, případně jeho pokynu k ukončení aplikace. Z tohoto stavu se aplikace může obnovit, což způsobí vyvolání metody `OnResume` nebo je ukončena, což ovšem nemusí být doprovázeno voláním speciální metody. Z tohoto důvodu je metoda `OnSleep` vhodná pro uložení stavu či nastavení aplikace. Metoda `OnSleep` není volána, pokud dojde k ukončení aplikace vlivem chyby, vytažení baterie uživatelem, ale i při ukončení ladícího módu ve Visual Studiu. Při volání `OnResume` operační systém automaticky obnoví stav a nastavení aplikace. Hodí se tedy například pro opětovné navázání spojení, které mohlo během neaktivity aplikace zrušeno.

MainPage.xaml.cs obsahuje pouze konstruktor, ve kterém volá metody `InitializeComponent`, pro načtení GUI z MainPage.xaml souboru. V tomto souboru je potom definice GUI pomocí jazyka xaml. GUI obsahuje instanci třídy `ContentPage` (dědící z třídy `Page`) a její obsah je nastaven na instanci třídy `Label`, kterému nastavuje vlastnosti `Text` (pro nastavení textu), `VerticalOptions` a `HorizontalOptions`, čímž jej zarovnává na střed obrazovky.

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="FirstApp.MainPage">
  <Label Text="Welcome to Xamarin Forms!"
          VerticalOptions="Center"
          HorizontalOptions="Center" />
</ContentPage>
```

Platformě specifické objekty se odkazují na `Xamarin.Forms.Platform.Android`, `Xamarin.Forms.Platform.iOS`, `Xamarin.Forms.Platform.WinRT`, `Xamarin.Forms.Platform.WinRT.Phone` nebo `Xamarin.Forms.Platform.WinRT.Tablet` v závislosti na platformě, na niž cílí. Jak již bylo uvedeno výše, tyto knihovny mají za úkol transformovat GUI elementy z knihovny Xamarin.Forms na platformě specifické objekty. Každá z těchto knihoven obsahuje statickou metodu `Forms.Init`, kterou je třeba volat při startu aplikace. Musí být také vytvořena instance třídy `App` z PCL knihovny.

Projekt pro iOS obsahuje třídu nazvanou `AppDelegate` dědící z `FromsApplicationDelegate`. V metodě `FinishedLaunching` potom volá metody `Init`, vytváří instanci `App` třídy z PCL, kterou předává jako parametr metodě `LoadApplication` definovanou v `FormsApplicationDelegate`.

```
using Foundation;
using UIKit;
namespace FirstApp.iOS
{
    [Register("AppDelegate")]
    public partial class AppDelegate :
          global::Xamarin.Forms.Platform.iOS.FormsApplicationDelegate {
   public override bool FinishedLaunching(UIApplication app, NSDictionary options)
   {
           global::Xamarin.Forms.Forms.Init();
           LoadApplication(new App());
            return base.FinishedLaunching(app, options);
    }
  }
}
```

Třída v projektu pro Android, MainActivity, je podobná jako u iOS, dědí však z `FormApplicationActivity` z knihovny `Xamarin.Forms.Platform.Android`.

```
    [Activity(Label = "PhotoViewer", Icon = "@drawable/icon",
Theme = "@style/MainTheme", MainLauncher = true,
ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
    public class MainActivity :
    global::Xamarin.Forms.Platform.Android.FormsAppCom patActivity
{
protected override void OnCreate(Bundle bundle)
       {
           mainActivity = this;
           TabLayoutResource = Resource.Layout.Tabbar;
           ToolbarResource = Resource.Layout.Toolbar;
           base.OnCreate(bundle);
           global::Xamarin.Forms.Forms.Init(this, bundle);
           LoadApplication(new App());
        }
    }
```

Aplikace pro Windows obsahují soubory App.xaml, App.xaml.cs, MainPage.xaml a MainPage.xaml.cs známé z rozboru sdílené knihovny. Metoda `OnLaunched` souboru App.xaml.cs volá `Xamarin.Forms.Forms.Init`. V MainPage.xaml.cs se opět volá platformě specifická metoda `LoadApplication` s parametrem objektu `App` z PCL.

```
    public sealed partial class MainPage
    {
        public MainPage()
        {
            this.InitializeComponent();
            LoadApplication(new PhotoViewer.App());
        }
    }
```

3 Visuální elementy a jazyk XAML

Uživatelské rozhraní pro Xamarin.Forms lze programovat v jazyce C# nebo v jazyce XAML, což se jeví jako lepší varianta. Kapitola vychází z [21][13][24].

3.1 Základy jazyka XAML

Jak již z názvu vyplývá, XAML je založen na formátu XML (*eXtensible Markup Language*). XAML nepodporuje smyčky či řízení toku programu. V jazyce XAML se třídy jazyka C# stanou XML elementy. Vlastnosti a další členy tříd se stanu XML atributy. Aby byla třída použitelná v jazyce XAML musí mít bezparametrický konstruktor (byť je možné specifikovat parametry konstruktoru pro speciální případy). Vlastnosti potom musí mít metodu `set` označenou modifikátorem public.

3.1.1 Překlad XAML kódu

Překladač jazyka XAML používá reflexi pro určení, zda existuje třída s názvem shodným s názvem elementu. Pokud ano, vytvoří novou instanci této třídy. Z množiny atributů zjistí, jaké vlastnosti byly nastaveny a pokusí se zadanou hodnotu převést na správný datový typ (jelikož v jazyk XAML není typovaný). Pokud je například atribut `IsVisible` nastaven na hodnotu `false`, XAML překladač ví, že správný datový typ je `Boolean`. Použije proto metodu `Boolean.Parse` a výsledek přiřadí vlastnosti `IsVisible` vytvořeného objektu.

XAML podporuje i výčtové typy za použití `Enum.Parse`. Pro komplexnější překlady lze využít atribut `TypeConverter` přiřazený typu nastavované vlastnosti. V tomto atributu je potom odkázáno na třídu dědící z abstraktní třídy `TypeConverter`. Ta definuje metody `CanConverFrom` a `ConvertFrom`. Pokud tedy překladač jazyka XAML pomocí reflexe zjistí, že datový typ nastavované vlastnosti má atribut `TypeConverter`, vytvoří instanci třídy definované v atributu, s jejíž pomocí (volání metody `ConvertFrom`) nastaví danou vlastnost. Například vlastnost `BackgroundColor`, definovaná v třídě `VisualElement` je typu `Color`, která obsahuje následující hlavičku:

```
[TypeConverter(typeof(Xamarin.Forms.ColorTypeConverter))]
    public struct Color
```

XAML překladač tedy vytvoří instanci třídy `ColorTypeConverter` a vlastnost typu `Color` nastavuje pomocí její metody `ConvertFrom`. Tím je umožněno definovat barvu v jazyce XAML různými způsoby (ve formátu RGG, ARGB a další).

Existuje i jiný způsob, jakým informovat překladač jazyka XAML o nutnosti konverze. Tímto způsobem je nastavení příslušného atributu přímo u nastavované vlastnosti. Tuto techniku využívá například vlastnost `FontSize`, která je typu `Double`. Není totiž možné připisovat atributy třídě `Double`. Zároveň je však vyžadováno přistupovat k této vlastnosti pomocí výčtu. Vlastnost `FontSize` je definována následovně:

```
[TypeConverter(typeof(FontSizeConverter))]
    public double FontSize { }
```

XAML umožňuje také nastavení více možných hodnot z výčtu současně (kdy se používá `Enum.Parse`). Je k tomu třeba použít atribut `Flags`. Hodnoty lze spojovat operátory logického součinu, součtu či exkluzivního součtu.

Pokud proběhne překlad kódu jazyka XAML úspěšně, je vygenerována třída v souboru s příponou xaml.g.cs. Tento soubor pak tvoří pár s xaml.cs souborem, proto je označen modifikátorem `partial`. V této třídě je definována metoda `InitializeComponent` a to následujícím způsobem:

```
private void InitializeComponent()
{
    this.LoadFromXaml(typeof(CodePlusXamlPage));
}
```

metoda `LoadFromXaml` je rozšiřující metoda třídy `View`. Jejím úkolem je vytvořit všechny GUI elementy a vložit je na stránku. Xamarin.Forms umožňuje kompilaci XAML kódu během překladu aplikace. Tohoto lze dosáhnout nastavením následujícího atributu.

```
[assembly: XamlCompilation(XamlCompilationOptions.Compile)]
```

`XamlCompilationOptions` má dva členy `Compile` a `Skip`. Nastavení atributu na úrovni sestavení ovlivní celý projekt. Pokud je třeba natavit `XamlCompilationOptions` v jednotlivých třídách lze toho dosáhnout přidáním atributu

```
[XamlCompilation(XamlCompilationOptions.Compile)]
```

do ovlivňované třídy.

3.1.2 Platformě specifické uživatelské rozhraní

Definice uživatelského rozhraní se může lišit v závislosti na platformě. V jazyce C# se používá metoda `Device.OnPlatform`. Jedná se o generickou metodu, jejíž návratový typ odpovídá generickému datovému typu. Tato metoda má čtyři parametry. První udává hodnotu pro zařízení iOS, druhý pro Android, třetí pro Windows a poslední pro ostatní zařízení. Parametry mají výchozí hodnotu `null` a jsou jmenné, takže lze nastavit hodnotu pouze pro určitou platformu na základě specifikace jména parametru. Následující kód nastavuje platformě specifickou barvu pozadí komponentě `Label`.

```
Label label = new Label();
            label.BackgroundColor = Device.OnPlatform<Color>
                 (Color.Red, Color.Black, Color.Blue);
```

Zařízení s operačním systémem iOS budou zobrazovat červené pozadí, v případě OS Android bude pozadí černé a zařízení s OS Windows nastaví pozadí modré. V knihovně Xamarin.Forms lze ovšem najít i generickou třídu s názvem `OnPlatform` s vlastnostmi generického typu nazvanými iOS, Android a WinPhone. Ekvivalent nastavení platformě specifických barev pozadí za použití této třídy je následující kód:

```
Label label = new Label();
           label.BackgroundColor = new OnPlatform<Color>
           {
               iOS = Color.Red,
               Android = Color.Black,
               WinPhone = Color.Blue
           };
```

V jazyce C# je doporučeno používat metodu `Device.OnPlatform` a třídu `OnPlatform` využívat v jazyce XAML. Pro specifikaci generického typu je v jazyce XAML použit atribut `x:TypeArguments`. Následující kód odpovídá výše uvedeným, ovšem při použití v jazyce XAML.

```
<Label>
   <Label.BackgroundColor>
      <OnPlatform x:TypeArguments="Color" iOS="Red"
                  Android="Black" WinPhone="Blue"/>
   </Label.BackgroundColor>
</Label>
```

Třída `OnIdiom` je podobná třídě `OnPlatform`, umožňuje však definici specifického uživatelského rozhraní v závislosti na typu zařízení (nastavením vlastnosti `Phone`, `Tablet` či `Desktop`).

Každá třída definující element uživatelského rozhraní má možnost definovat takzvanou výchozí vlastnost. Taková třída musí mít jednu z vlastností (výchozí vlastnost) označenu speciálním atributem. Výchozí vlastnosti jsou přiřazeny hodnoty mezi začínajícím a ukončujícím XML tagem jazyka XAML. Třída `Layout<T>` má například výchozí vlastnost `Children`, což dokazuje hlavička této třídy.

```
[Xamarin.Forms.ContentProperty("Children")]
public abstract class Layout<T> : Layout, IViewContainer<T>
```

3.1.3 Parametry metod a konstruktorů v jazyce XAML

Objekty uživatelského rozhraní běžně vznikají voláním bezparametrového konstruktoru. Existuje ovšem i způsob volání konstruktoru parametrického a to pomocí elementu `x:Arguments`. XAML dokonce umožňuje specifikovat základní datové typy jazyka C#, které lze využít ke specifikaci typu parametrů. Definici typu umožňuje jmenný prostor

`http://schemas.microsoft.com/winfx/2009/xaml`. Mezi definovatelné typu patří `x:Object`, `x:Int32`,`x:Char`, `x:Array` a další.

Dalším zajímavým elementem je element `x:FactoryMethod`, který zpřístupňuje metody jazyka C# v jazyce XAML. Taková metoda musí být statická a veřejná. Parametry jí lze předat pomocí `x:Atributes`.

3.1.4 Přístup ke XAML elementům

Často je třeba přistoupit k elementům uživatelského rozhraní z kódu na pozadí. První, ne příliš dobrou metodou je procházení visuálního stromu, kdy kód na pozadí zná vztahy mezi GUI elementy. Může potom procházet strom od kořene a libovolný z elementů získat, jelikož tyto elementy jsou uloženy ve vlastnostech elementů rodičovských. Lepší variantou je používat v jazyce XAML atribut `x:Name`. Pokud je definován atribut `x:Name` GUI elementu, může k němu kód na pozadí (případně i samotný jazyk XAML) pomocí tohoto jména přistupovat, jako by byl jeho datovým členem. Tento přístup je možný z toho důvodu, že je tyto objekty se stanou privátními proměnnými třídy definované ve vygenerovaném v souboru s příponou xaml.g.cs. Jelikož při vytváření těchto objektů dochází k jejich hledání pomocí jména, musí být jména v rámci jednoho XAML souboru unikátní.

3.1.5 Rozpis vlastností v jazyce XAML

Některé vlastnosti jazyka C# mohou obsahovat komplexní objekty. Příkladem je vlastnost `Children` třídy `StackLayout`. Jedná se o kolekci objektů dědících z třídy `View`. V takovémto případě lze použít následující syntaxi:

```
<StackLayout>
    <StackLayout.Children>
        <Button Text="klikni"></Button>
        <Label BackgroundColor ="Red"/>
    </StackLayout.Children>
</StackLayout>
```

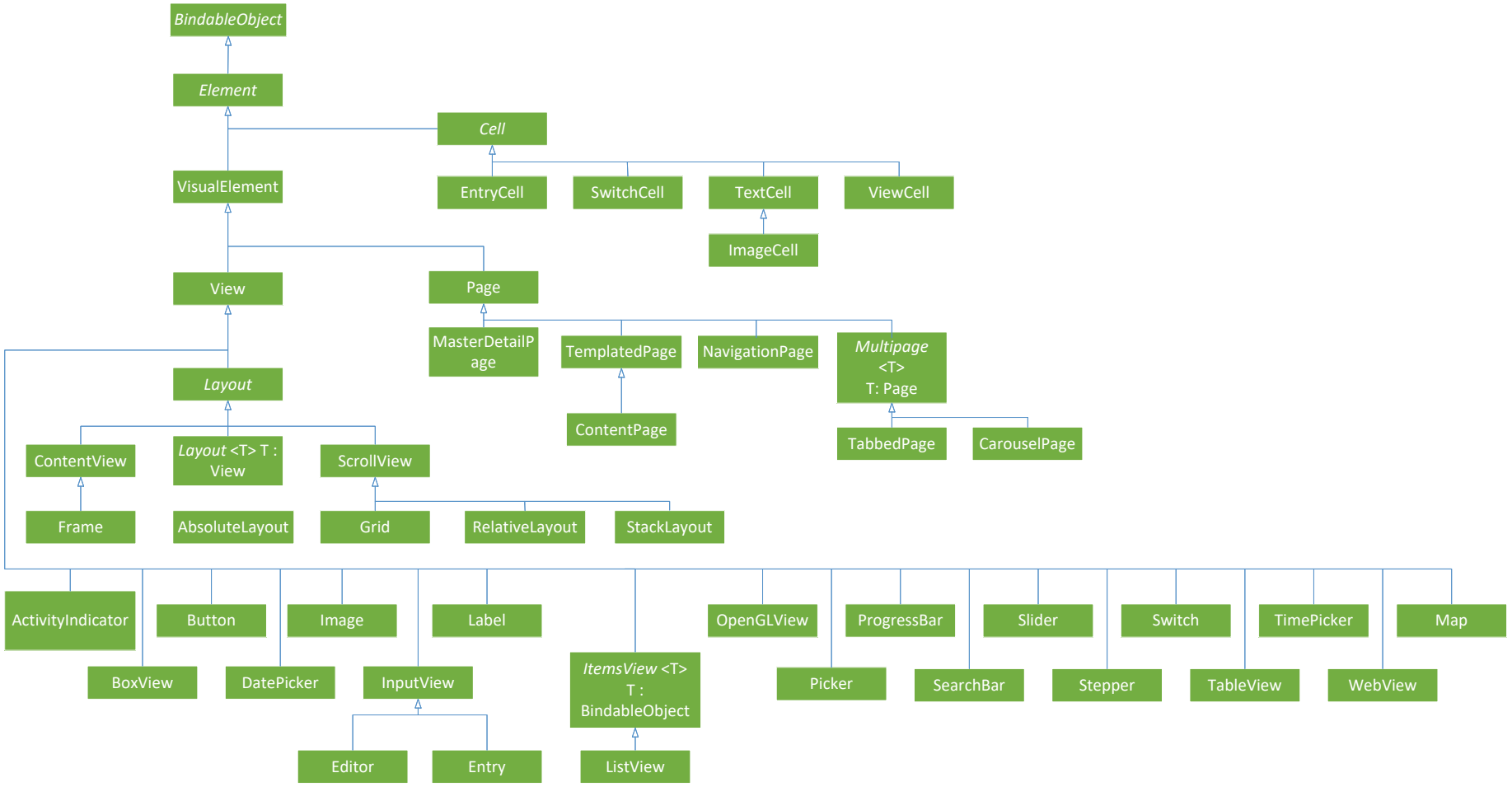
Vlastnost `StackLayout.Children` je nazývána jako property element. Syntaxe vzniká spojením názvu třídy a její vlastnosti tečkou. Tímto způsobem lze přepsat celý kód. Výsledek je následující:

```
<StackLayout>
    <StackLayout.Children>
        <Button>
            <Button.Text>klikni</Button.Text>
        </Button>
        <Label>
            <Label.BackgroundColor>Red </Label.BackgroundColor>
        </Label>
    </StackLayout.Children>
</StackLayout>
```

3.2 Visuální elementy

Objekty uživatelského rozhraní jsou pojmenovány různě na různých platformách (controls pro Windows, widget pro Android a view pro iOS). V Xamarin.Forms jsou nazývány visuální elementy (dědící z třídy `VisualElement`). Visuální elementy lze rozdělit do tří supin. Stránky (dědící z třídy `Page`), kterých může aplikace obsahovat i více. V nich jsou obvykle umístěny layouty (dědící z třídy `Layout`), jejichž úkolem je uspořádat poslední skupinu dědící z třídy `View` (například `Slider`, `Label`, `Button` a další), či další elementy dědící z třídy `Layout`, dle specifických pravidel. Visuální elementy jsou řazeny ve formátu rodič- potomek, kdy pro některé rodiče je možná i varianta s více potomky.

Pro tvorbu uživatelského rozhraní jsou důležité i další třídy dědící ze třídy `Element`. Hierarchii tříd ze jmenného prostoru Xamarin.Forms sloužících k zobrazení visuálních zobrazuje Obr. 3.



Obr. 3 Hierarchie dědičnosti visuálních elementů

3.2.1 Třída Element

Důležitou vlastností třídy `Element` je `StyleId`, kterou lze využít k uchování informací o GUI elementu. Vlastnost `Parent` slouží k definován rodiče a `ClassId` sloužící k definici sémanticky podobných elementů.

3.2.2 Třída VisualElement

Třída `VisualElement` definuje mnoho vlastností, které lze rozdělit do několika skupin.

První jsou vlastnosti spojené s animacemi a transformacemi. Patří sem `AnchorX`, `AnchorY` udávající středový bod transformací či animací, vlastnost `RotationX` a `RotationY` udávající úhel o který je element otočen kolem osy X nebo Y. Podobně `TranslationX` a `TranslationY` udávají posun elementu ve směru osy X nebo Y. Poslední vlastností tohoto typu je `Scale` udávající hodnotu násobku zvětšení elementu.

Druhá skupina zahrnuje vlastnosti spojené s polohou. `Height` a `Width` udávající výšku a šířku elementu, jsou pouze pro čtení. Požadovaná výška a šířka se nastavuje pomocí `HeightRequest` a `WidthRequest`. Pouze pro čtení jsou také označené vlastnosti `X` a `Y` udávající pozici levého horního rohu elementu.

Třetí skupina definuje vzhled elementu. Patří sem:

- `IsVisible` udávající, zda je element viditelný
- `IsEnabled` pro nastavení možnosti/nemožnost interakce uživatele s elementem
- `IsFocused` pouze pro čtení. Udává, zda byl element označen uživatelem
- `BackgroundColor` pro nastavení barvy pozadí
- `Opacity` určující průhlednost elementu
- `Bounds`, pro získání hranice elementu (pouze ke čtení)
- `Style` definující nastavený styl.

Dalšími vlastnostmi pouze pro čtení jsou `Navigation` pro informaci o navigaci mezi elementy, `Triggers` obsahující instance třídy `TriggerBase` přidané k elementu a `Behaviours` obsahující instance třídy `Behaviour` přidané k elementu. Poslední vlastnost je `Resources`, umožňující definovat vlastní nastavení platné v rámci elementu. Funkce mnoha z těchto vlastností bude rozebrána níže.

3.2.3 Třída View

Tato třída rozšiřuje `VisualElement` o důležité vlastnosti. Dvě z nich, `HoriznotalOptions` a `VerticalOptions`, specifikují pozici elementu v rodiči. Vlastnost

`Margin` udává odsazení elementu od ostatních elementů nebo od kraje stránky. Kolekce `GestureRecognizers` je pouze pro čtení a rozšiřuje schopnosti elementu v oblasti interakce s uživatelem. Z třídy `View` dědí mnoho tříd. Níže budou všechny rozebrány.

Visuální elementy lze dělit na interaktivní a presentační. Pomocí interaktivních elementů předává data uživatel aplikaci, zatímco presentační slouží k zobrazení informace uživateli.

3.3 Interaktivní visuální elementy

3.3.1 Tlačítko

Nejvýznamnějším interaktivním elementem je tlačítko (třída `Button`). Tlačítko může být zobrazeno pouze s textem nebo i s obrázkem. Třída obsahuje mnoho vlastností pro změnu výchozího vzhledu. Jedná se o vlastnosti `FontFamily`, `FontSize`, `FontAtributes`, `TextColor`, `BorderColor`, `BorderWidth`, `BorderRadius` a `Image`. Jako všechny ostatní třídy pro definici UI dědí `Button` vlastnost `BackgroundColor` třídy `VisualElement` a vlastnosti `HorizontalOptions` a `VerticalOptions` z třídy `View`. Některé z těchto vlastností fungují odlišně v závislosti na platformě. Například nastavení vlastnosti `BorderWidth` na nenulovou hodnotu způsobí zobrazení černého okraje pouze na iOS zařízeních.

Nastavení pouze vlastnosti `BorderColor` způsobí změnu pouze u Windows zařízení. Nastavením obou výše uvedených vlastností lze dosáhnout zobrazení barevného rámečku na všech platformách. Vlastnost `BorderRadius` je funkční na zařízeních s OS Android a iOS, pokud je viditelný rámeček. V prostředí Windows 10 je tato vlastnost nefunkční a při použití ve Windows 8.1 není zaobleno pozadí, pokud je nastavena vlastnost `BackgroundColor` na hodnotu jinou než výchozí.

Nejdůležitějším datovým členem jsou ovšem ty, které jsou spojené se stiskem tlačítka. Jedná se o událost `OnClick` vyvolanou při stisku tlačítka a vlastnost `Command` typu `ICommand`, která je společně s vlastností `CommnadParameter` využívána k ošetření stisku tlačítka
při použití návrhového vzoru MVVM.

Mimo tlačítko definuje Xamarin.Forms další interaktivní UI elementy. Jedná se o `Slider`, `Stepper`, `Switch`, `Entry`, `SearBar`, `Editor`, `TimePicker` a `DatePicker`.

3.3.2 Switch

Switch slouží k nastavení hodnoty typu `Boolean` uživatelem. Stejnojmenná třída definuje vlastnost `IsToggled` typu `Boolean`. Její změna způsobí vznik události `Toggled`.

3.3.3 Elementy pro nastavení data a času

Jak již název napovídá, umožňují komponenty `DatePicker` a `TimePicker` nastavení data a času. `DatePicker` obsahuje čtyři vlastnosti: `MinimumDate` (výchozí hodnota je 1.1.1900), `Date` (výchozí hodnota je `Datetime.Today`), `MaxDate` (výchozí hodnota je 31.12.2100) a `Format` specifikující formát data. Time picker definuje pouze dvě vlastnosti `Time` a `Format`. Vlastnost `Time` je typu `TimeSpan` a indikuje dobu uplynulou od půlnoci. Při změně vlastnosti `Time` nedojde k vyvolání žádné události.

3.3.4 Elementy pro zadávání textu

Elementy `Entry`, `Editor` a `SearchBar` umožňují uživateli zadávat text. `Entry` a `Editor` dědí z třídy `InputView`, zatímco `SearBar` dědí z třídy `View`. V prostředí mobilních aplikací je běžné zadávat hodnoty pomocí virtuální klávesnice. Jak bylo uvedeno výše, třída `Element` definuje vlastnost `IsFocused`. Pokud je tato vlastnost instance typu `Entry`, `Editor` nebo `SeachBar` nastavena na hodnotu `true` (změnu indikuje událost `Focused`), zobrazí se virtuální klávesnice (je-li na zařízení dostupná). Po jejím nastavení na hodnotu `false` (změnu indikuje událost `Unfocused`) klávesnice zmizí. Virtuální klávesnici lze měnit tak, aby odpovídala charakteru vkládaného textu. K tomuto účelu slouží vlastnost `Keyboard` typu `Keyboard` třídy `InputView`. Třída `Keyboard` definuje sedm statických vlastností jen pro čtení: `Chat`, `Default`, `Email`, `Numeric`, `Telephone`, `Text` a `Uri`.

Třída `Entry` definuje čtyři vlastnosti. Vlastnost `IsPassword` typu `Boolean`, která při nastavení na hodnotu true nahradí znaky jinými zástupnými znaky. Vlastnost `Text` typu `string` s hodnotou textu zobrazeného pomocí elementu, `TextColor` typu `Color` pro nastavení barvy textu a `Placeholder` typu `string` obsahující text zobrazený na pozadí elementu v době, kdy vlastnost `Text` obsahuje prázdný řetězec. Změna vlastnosti `Text` vyvolá událost `TextChanged`. Další událostí je `Completed` vyvolaná v době, kdy uživatel stiskne klávesu, kterou oznamuje, že ukončil psaní.

Třída `Editor` je zaměřena na zadávání víceřádkového textu. S třídou `Entry` má společného rodiče a definuje i stejné události (`TextChange` a `Completed`).

`SearchBar` je podobný jako `Entry`, nedědí však ze třídy `ImputView` a nemá tedy vlastnost `Keyboard`. Navíc má dvě tlačítka (platformě specifické). První z nich slouží k vyhledávání v textu, druhé potom ke smazání textu. Třída definuje pět vlastností: `Text` a `PlacehoderText`, které fungují stejně jako u elementu `Entry`, `CancelButtonColor` typu `Color`, `SearchCommand` a `SearchCommandParameter` využívané pro detekci stisku tlačítka pomocí data bindingu.

3.3.5 Elementy pro zadávání číselných hodnot

Poslední dva interaktivní elementy umožňují uživateli zadávat numerické hodnoty. Jedná se o `Slider` a `Stepper`. `Slider` umožnuje nastavit hodnotu mezi minimem stanoveným vlastností `Minimum` (s výchozí hodnotou 0) typu `Double` a maximem (s výchozí hodnotou 1) stanoveným vlastností `Maximum` stejného typu. Jeho poslední vlastností je `Value` udávající nastavenou hodnotu typu `Double`. Její změna vyvolá událost `ValueChanged`. Tato událost obsahuje dva parametry. Prvním z nich je typu `Object` udávající element, který událost vyvolal a druhý je typu `ValueChangedEventArgs` a obsahuje vlastnosti `OldValue` a `NewValue`, obsahují hodnotu před změnou a po změně.

Pro hodnoty `Maximum` a `Minimum` musí vždy platit, že `Maximum` je větší než `Minimum`. Z tohoto důvodu je třeba dbát na pořadí definování těchto vlastností. Pokud bude nejprve definována vlastnost `Minimum` na hodnotu větší nebo rovnu jedné (výchozí hodnota vlastnosti `Maximum`), bude vyhozena výjimka.

`Stepper` má podobný význam jako `Slider`, vypadá však zcela jinak. Skládá se ze dvou tlačítek sloužících k inkrementaci a dekrementaci hodnoty. `Stepper` má všechny vlastnosti i událost zmíněné u třídy `Slider`. Výchozí hodnota vlastnosti `Maximum` je ovšem 100. Navíc definuje vlastnost `Increment` (s výchozí hodnotou 1), která udává hodnotu, která bude přičtena nebo odečtena při stisku inkrementačního nebo dekrementačního tlačítka. K inkrementaci či dekrementaci dochází po celou dobu, po kterou uživatel udržuje tlačítko stisknuté.

3.4 Presentační visuální elementy

3.4.1 Element Label

Element `Label` je obsažen ve výchozím projektu. Jeho úkolem je zobrazování textu. Vlastnosti třídy `Label` jsou povětšinou spojené s formátováním textu. Pomocí `LineBreakMode` lze nastavit jakým způsobem bude zalamován text. `FontSize` určuje velikost textu. Vlastnost `Text` obsahuje vlastní text, `TextColor` jeho barvu, `FontSize` velikost a `FontFamily` písmo. Vlastnost `FontSize` lze nastavit hodnotou číselnou nebo výčtovou typu `NamedSize`. `FontAtributes` umožnuje nastavit písmo tučně, kurzívou, obojím či ani jedním. Vlastnost `FormattedText` typu `FormattedString` obsahuje kolekci objektů typu `Span`. Vložením prvků do této kolekce lze formátovat jednotlivé části textu. Vlastnosti `VerticalTextAlignment` a `HorizontalTextAlignment` jsou typu `TextAlignment` a slouží k zarovnání textu. Výčet `TextAlignment` sestává z hodnot

`Center`, `End` a `Start`. Hodnota `Start` značí levý (pro `HorizontalTextAlignment`) a horní (`VerticalTextAlignment`) okraj elementu.

3.4.2 Element Image

Těžko si lze představit mobilní zařízení bez obrázků. K jejich zobrazení slouží třída `Image`. Tato třída obsahuje vlastnost `Source` typu `ImageSource` definující soubor obsahující bitmapu, `Aspect` udává způsob roztažení obrázku, `IsOpaque` udává, zda má být obrázek průhledný a `IsLoading` indikuje, zda byl již obrázek načten. Objekt typu `ImageSource` lze získat několika způsoby. Metoda `ImageSource.FromUri` slouží k zobrazení bitmapy z internetu, další možností je `ImageSource.FromResource` používaná, je-li soubor přiřazen jako přiložený (embeded) soubor PCL projektu. `ImageSource.FromFile` slouží k načtení z platformě specifických projektů a `ImageSource.FromStream` slouží k načtení obrázku z objektu typu `Stream`. První, třetí a čtvrtou metodu lze nahradit pomocí tří potomků třídy `ImageSource`. Jedná se o `UriImageSource`, `FileImageSource` a `StreamImageSource`. Použití metod je jazyce C# jednodušší, třídy mohou být ovšem potřebné pro jazyk XAML. `UriImageSource` definuje vlastnost `CachingEnabled` (výchozí hodnota je `true`), po jejímž nastavení jsou bitmapy ukládány do privátního úložiště aplikace. Pomocí vlastnosti `CachingValidity` lze nastavit dobu platnosti takových souborů (výchozí je jeden den). Vlastnost `Aspect` stejnojmenného výčtového typu lze nastavit na hodnoty `AspectFit` (výchozí hodnota), `Fill` a `AspectFill`. Při nastavení `AspectFit` bude bitmapa roztažena na velikost svého rodiče, zachová si ovšem poměr stran (na okrajích rodiče může být volné místo). Nastavení hodnoty `Fill` způsobí roztažení bitmapy na velikost rodiče a poměr stran není zachován. Poslední možnost `AspectFill` spojuje obě více uvedené. Poměry stran zůstanou stejné a přesahující část bitmapy je oříznutá. Zařízení s OS Android a iOS ponechávají středovou část bitmapy, zatímco zařízení s OS Windows ponechají její část začínající horním levým rohem.

K roztažení na velikost rodiče nedojde v případě, že bitmapa je menší než rodič a zároveň vlastnosti `VerticalOptions` a `HorizontalOptions` elementu `Image` nejsou nastaveny na hodnotu `Fill`. K roztažení také nedojde, je-li element `Image` obsažen v elementu `StackLayout`. Pokud nedochází k roztažení bitmapy, je ovšem zobrazena v jiné velikosti v závislosti na platformě. Zařízení s OS Android a iOS zobrazují bitmapu tak, že jeden pixel bitmapy přiřadí jednomu pixelu zařízení. To způsobí různé velikosti bitmap v závislosti na hustotě pixelů zařízení. Na platformě Windows bude bitmapa zobrazena v abstraktních jednotkách DIU (*Device Independent Unit*).

V příkladech uvedených výše se často pracovalo se strukturou `Color`. Hodnotu lze vytvořit buď z barevných složek formátu RGB, RGBA, HSL, HSLA nebo pomocí statických

hodnot definovaných ve struktuře `Color`. Zajímavými hodnotami jsou hodnoty `Color.Default` (s RGB hodnotami nastavenými na -1) a `Color.Accent`. Hodnota `Default` je rozdílná pro různé elementy, i pro různé platformy. Příkladem je třída `Label`. Hodnota `Default` vlastnosti `BackgroundColor` odpovídá hodnotě `Color.Transparent` zatímco u vlastnosti `TextColor` je výchozí barva závislá na platformě (černá pro iOS a UWP a bílá jinde). V prostředí Android a iOS platí, že `Color.Accent` je taková barva, která je dobře viditelná při proti výchozímu pozadí. V prostředí Windows je tato hodnota zvolena uživatelem.

3.4.3 Elementy indikující práci na pozadí

`ActivityIndicator` má dvě vlastnostmi `IsRunning` a `Color`. Tato komponenta slouží k informování uživatele, o probíhající činnosti na pozadí.

Podobný význam má `ProgressBar`. Vzhledově je podobný elementu `Slider` a informuje uživatele o stavu operace na pozadí. Vlastnost `Progress` typu `Double` s hodnotou 0 až 1 definuje poměrnou část komponenty, jejíž zbarvení indikuje stav prováděné činnosti.

3.4.4 Element BoxView

Pro zobrazení barevného obdélníku slouží element `BoxView`. Jeho jedinou vlastností je `Color` typu `Color`. Ve výchozím nastavení má velikost 40 (pokud nejsou jeho vlastnosti `HorizontalOptions` či `VerticalOptions` nastaveny na hodnotu `Fill`).

3.4.5 Element WebView

Komponenta `WebView` slouží k zobrazení HTML souborů. Její metody `GoBack` a `GoForward` umožnují pohyb mezi stránkami. Vlastnosti `CanGoBack` a `CanGoForward` typu `Boolean` lze využít ke zjištění, zda má smysl metody volat. Další vlastností je `Source` typu `WebViewSource` specifikující webovou stránku. Začátek navigace na novou stánku doprovází událost `Navigating` a po ukončení navigace je vyvolána událost `Navigated`.

3.4.6 Element OpenGLView

Komponenta `OpenGLView` umožňuje zobrazit grafické komponenty a to pomocí 2D (*Two-Dimensional*) i 3D (*Three-Dimensional*) grafiky pomocí knihovny OpenGL. Její vlastnosti jsou `HasRenderLoop` a `OnDisplay`.

3.4.7 Element Map

Xamarin.Forms obsahuje speciální element pro zobrazení mapy jménem `Map`. Ten má celkem šest vlastností. Tři jsou typu `Boolean`. Konkrétně `HasScrollEnabled` určující, zda lze s mapou pohybovat, `HasZoomEnabled` určující, zda lze mapu přibližovat a `IsShowingUser` určující, zda je na mapě zobrazena současná poloha uživatele. `MapType` stejnojmenného typu určuje typ mapy. Poslední dvě vlastnosti jsou pouze pro čtení. Vlastnost `Pins` typu `IList<Xamarin.Forms.Maps.Pin>` specifikuje zvýrazněné body na mapě a `VisibleRegion` typu `MapSpan` určuje, jaká část mapy je viditelná.

3.5 Visuální elementy pro zobrazení kolekcí

Zbývající tři elementy jsou `ListView` dědící z `ItemView` , `Picker` a `TableView`. Ty slouží k zobrazení kolekce položek. Tyto elementy mohou být z principu presentační i interaktivní. Všechny umožňují rolování obrazovky.

3.5.1 Element Picker

Pomocí elementu `Picker` může uživatel vybírat jednu položku typu `string` z množiny hodnot. Tato množina je definována vlastností `Items` typu `IList<string>`. Index vybraného řetěze je uložen ve vlastnosti `SelectedIndex`. Element umožnuje nastavení titulku, který se zobrazí nad ním a to pomocí vlastnosti `Title`.

3.5.2 Třída Cell a její potomci

`ListView` a `TableView` využívají k zobrazení kolekce dědice třídy `Cell`. Třída `Cell` je abstraktní třída dědící z třídy `Element` ne však z třídy `VisualElement`. Nejedná se tedy o element zobrazitelný samostatně v uživatelském rozhraní. Třída definuje pět veřejných vlastností. Vlastnost `Heigth` určující výšku, vlastnost `IsEnabled` určující, zda lze s touto komponentou interagovat a tři vlastnosti pouze pro čtení. První z nich, `RenderHeight`, udává skutečnou výšku. Další dvě jsou spojeny s kontextovým menu, které může být u každé třídy dědící z třídy cell zobrazeno. První z vlastností je `HasContentActions` typu `Boolean` udávající, zda kontextové menu obsahuje alespoň jeden prvek. Vlastní prvky potom definuje vlastnost `ContextActions` typu `IList<MenuItem>`. Třída `MenuItem` definuje vlastnosti `Text` typu `string` pro zobrazení popisku, `Icon` typu `FileImageSource` pro zobrazení ikony, `IsDestruktice` určující, zda tato položka maže objekt třídy `Cell` ke kterému je přiřazena a vlastnosti `Command` a `CommandParameter` pro obsluho stisknutí za využití data bindingu. Třída definuje také událost `Clicked` vyvolanou při stisknu položky. Přímými

potomky třídy `Cell` jsou `EntryCell`, `SwitchCell`, `ViewCell` a `TextCell`, která je rodičem pro `ImageCell`.

Třída `TextCell` slouží pro zobrazení textu, s možností rozdělení textu na hlavní a detailní. Definuje šest vlastností. Vlastnosti `Text` a `Detail` typu `string` obsahují zobrazovaný text (hlavní a detailní). Barvu těchto textů definují vlastnosti `TextColor` a `DetailColor`. Třída obsahuje možnost obsluhy stisku pomocí vlastností `Command` a `CommandParameter`.

Z `TextCell` dědí `ImageCell`, rozšiřující její funkcionalitu o možnost přidání obrázku pomocí vlastnosti `ImageSource` typu `ImageSource`, který byl popsán v souvislosti s elementem Image.

Dalo by se říci, že třída `SwitchCell` sestává z elementů `Label` a `Switch`. Umožňuje tedy vyžádat od uživatele informaci typu `Boolean` na základě určitého textu. Definuje vlastnosti `On` typu `Boolean`, určující stav obsaženého elementu `Switch` a vlastnost `Text` typu `string` uchovávající doprovodný text.

Stejně jako `SwitchCell` obsahoval element `Switch`, obsahuje `EntryCell` element `Entry`. Doprovodný text zůstává. Tento doprovodný text lze nastavit pomocí vlastnosti `Label` typu `string` a jeho barvu pomocí `LabelColor` typu `Color`. Další vlastnosti souvisí s textem zadávaným uživatelem. Tyto vlastnosti mají shodný název i význam při použití v elementu `Entry`, proto o nich zde nebude dále pojednáno. Jedná se o `Text`, `Placeholder`, `Keyboard`, a `HorizontalTextAlignment`.

Třída `ViewCell` umožňuje definovat libovolný design buňky pomocí vlastnosti `View` stejnojmenného typu.

3.5.3 Element ListView

`ListView` slouží k zobrazení kolekce prvků stejného typu. Umožňuje také výběr jednotlivých prvků jejich stiskem. Rodičem třídy `ListView` je generická třída `ItemsView<T>`, která dědí z třídy `View`. `ItemsView<T>` definuje dvě vlastnosti. `ItemsSource` typu `IEnumerable` slouží k definici kolekce, která bude zobrazena. `ItemTemplate` typu `DataTemplate` určuje vzhled jednotlivých prvků kolekce zobrazených v `ListView`. Samotná třída `ListView` obsahuje šestnáct vlastností, které lze rozdělit do několika skupin.

- **Vlastnost umožňující interakci s elementem**
  Vlastnost `SelectedItem` obsahuje objekt, který byl zobrazen v tom prvku, který uživatel stiskl.

- **Vlastnosti definující rozměry jednotlivých prvků**

  Všechny položky mohou být zobrazeny se stejnou či různou velikostí. Rozhoduje o tom vlastnost `HasUnevenRows` typu `Boolean`. Vlastní výšku lze nastavit pomocí vlastnosti `RowHeight`. Ve výchozím nastavení jsou prvky odděleny dodatečným místem (kromě platformy Windows). Viditelnost tohoto místa je dána vlastností `SeparatorVisibility` stejnojmenného výčtového typu (obsahující možnosti `Default` a `None`). Barvu tohoto oddělovače určuje `SeparatorColor` typu `Color`. Oddělovač není funkční na platformě Windows.

- **Vlastnosti umožňující seskupování položek**

  Někdy je vhodné seskupit položky ať už dle nějaké společné vlastnosti nebo dle abecedy. Pro dosažení tohoto cíle je třeba nastavit vlastnost `IsGrupingEnabled` na hodnotu `true`. Vlastnosti `ItemsSource` se potom přiřadí kolekce, kde každý její prvek obsahuje kolekci (kolekce kolekcí). I v tomto případě je však vybraný prvek (vlastnost `SelectedItem`) typu obsaženého ve vnitřní kolekci. Vlastnost `GroupDisplayBinding` slouží k zobrazení titulku skupiny a `GroupShortNameBinding` k zobrazení krátkého popisu skupiny pro vyhledávání. Těmto vlastnostem se přiřadí vlastnost objektu definovaného v kolekci `ItemsSource`. `GroupHeaderTemplate` typu `DataTemplate` umožňuje vytvoření vlastního vzhledu oddělovačů skupin.

- **Vlastnosti spojené s načítáním elementů**

  Rolování elementem `ListView` vyžaduje jistý čas pro vytvoření nových prvků k zobrazení. Načítání je doprovázeno metodou `Execute`, kterou definuje vlastnost `RefreshCommand` typu `ICommand`. Při načítání je také vlastnost `IsRefreshing` nastavena na hodnotu `true`. Vlastnost `IsPullToRefreshEnabled` určuje, zda je uživateli umožněno rolování provést.

- **Vlastnosti související s hlavičkou a zápatím**

  `ListView` umožňuje specifikovat hlavičku a zápatí, které budou stále zobrazeny v horní či spodní části obrazovky bez ohledu na to, zda uživatel roluje obsahem elementu. Vlastnost `Header` definuje hlavičku a vlastnost `Footer` zápatí. Obě vlastnosti jsou typu `object` a pro zobrazení volají metodu `ToString`. Visuální podobu záhlaví a zápatí lze změnit pomocí `FooterTemplate` a `HeaderTemplate` typu `DataTemplate`.

3.5.4 Element TableView

`TableView` stejně jako `ListView` zobrazuje instance typu `Cell` v řádcích. `ListView` je ovšem vhodný zejména pro zobrazení kolekcí obsahující prvky stejného typu. `TableView` většinou zobrazuje jeden objekt, jehož vlastnosti zobrazuje pomocí více instancí typu `Cell`.

`TableView` definuje čtyři vlastnosti, z nichž dvě (`RowHeight` a `HasUnevenRows`) mají stejný význam jako u `ListView`. Výchozí vlastností je vlastnost `Root` typu `TableRoot` umožňující definovat kořen prvků zobrazených v `TableView` (objekty typu `Cell` mohou být hierarchicky uspořádány). Vlastnosti `Root` je možné přiřadit kolekci, jejíž každý prvek obsahuje kolekci prvků typu `Cell`. Popis principu následuje. Třída `TableRoot` dědí z generické třídy `TableSectionBase<TableSection>`, která mimo jiné implementuje `IEnumerable<T>` a `IList<T>`. Rodičem této generické třídy je stejnojmenná negenerická třída s vlastností `Title` typu `string`, přiřazující titulek každé ze sekcí. `TableSection` je typu `TableSectionBase<Cell>`. Vlastnost `Root` tedy obsahuje kolekci typu `TableSection`, kdy každý prvek z této kolekce obsahuje kolekci prvků typu `Cell`. `TableSection<T>` umožňuje přidávání elementů do kolekce pomocí metody `Add` s argumentem `T` nebo `IEnumerable<T>`.

Poslední vlastností je `Intent` výčtového typu `TableIntent`. Tato vlastnost slouží pro informování rendererů (objektů vytvářejících platformě specifické UI) o účelu použití daného objektu `TableView`. Tento výčet obsahuje prvky `Data` (`TableView` slouží pro zobrazení dat podobného typu), `Form` (`TableView` má formát formuláře), `Menu` (`TableView` slouží pro zobrazení menu) a `Settings` (`TableView` obsahuje především elementy `Switch`, `Label` a další).

3.6 Rozšíření jazyka XAML

Často využívané je rozšíření jazyka XAML. Výše bylo zmíněno, že hodnoty vlastnosti lze v jazyce XAML nastavovat pomocí textu s možnou podporou třídy `TypeConverter`. Další možností bylo definování hodnot pomocí jmenného prostoru `http://schemas.microsoft.com/winfx/2009/xaml` běžně označovaného `x` (`x:Int32` , `x:Array` a další). Obdobou těchto přístupů v jazyce C# pro datový typ `int` je `int x =5` (obdoba použité `x: Int32`) a `Int32.Parse("5")`, což je obdobou použití `TypeConverteru`. V jazyce C# lze ovšem přiřadit hodnotu z jiné proměnné, případně jako výsledek volání metod. Tuto funkcionalitu nahrazuje právě rozšíření jazyka XAML. Jedná se o třídy implementující rozhraní `IMarkupExtension`. Toto rozhraní definuje metody `ProvideValue` s parametrem typu `IServiceProvider` a návratovou hodnotou `object`.

Tato metoda zprostředkovává hodnotu jazyku XAML. Rozhraní `IServiceProvider` definuje metodu `GetService` s parametrem typu `Type` a návratovým typem `object`.

Lze vytvářet vlastní třídy implementující `IMarkupExtension`, některé jsou ovšem již připraveny ve specifikaci XAML 2009 (bývají označeny prefixem x). Patří sem `x:Static`, `x:Reference`, `x:Type`, `x:Null` a `x:Array`. Jsou implementovány v třídách s názvem shodným s hodnotou za x: spojeným se slovem „Extension“ (`StaticExtension`, `TypeExtension` atd.) ve jmenném prostou `Xamarin.Forms.Xaml`. V tomto jmenném prostoru jsou také třídy `StaticResourceExtension`, `DynamicResourceExtension` a `BindingExtension`. Ty zastřešují XAML rozšíření `StaticResource`, `DynamicResource` a `Binding`, které vznikly v technologií WPF (*Windows Presentation Fundation*). Ve jmenném prostoru `Xamarin.Forms` je třída `ConstraintExpression`. Jedná se o jedinou rozšiřující třídu unikátní pro Xamarin využívané v elementu `RelativeLayout`.

3.6.1 Rozšíření x:Static

`x:Static` slouží ke zpřístupnění statických vlastností či výčtových typů. Třída `StaticExtension` definuje jednoho člena, vlastnost `Member` typu `string`, kterou je třeba nastavit na požadovanou hodnotu. Například vlastnost `HorizontalOptions` je typu `LayoutOptions` což je typ výčtový.

```
<Slider HorizontalOptions="Center"></Slider>
```

lze tedy přepsat pomocí `x:Static` následovně:

```
<Slider>
    <Slider.HorizontalOptions>
        <x:Static Member="LayoutOptions.Center"></x:Static>
    </Slider.HorizontalOptions>
</Slider>
```

Tento zápis lze zkrátit pomocí složených závorek.

```
<Slider HorizontalOptions="{x:Static Member = LayoutOptions.Center}"></Slider>
```

Jelikož vlastnost `Member` je vlastností výchozí, lze zápis dále zkrátit.

```
<Slider HorizontalOptions="{x:Static LayoutOptions.Center}"></Slider>
```

Z výše uvedených příkladů by se mohlo zdát, že `x:Static` kód spíše prodlužuje. S tímto přístupem lze však zpřístupnit vlastní konstanty, výčtové typy či statické proměnné, které lze následně měnit pouze na jednom místě a ovlivňovat více elementů. Příkladem může být definování hodnoty vlastnosti `Margin`, sdílené některými komponentami a to například takto:

```
public static class Constants
{
    public static Thickness MarginValue = 5;
}
```

V jazyce XAML lze potom definovat:

```
<Label Margin = "{x:Static local:Constants.MarginValue}"/>
```

3.6.2 Rozšíření StaticResource

Toto rozšíření se od `x:Static` liší. Umožňuje přiřazovat vlastnostem objekty definované v kolekci `ResourceDictionary` (zatímco `x:Static` přiřazovalo pouze statické proměnné a vlastnosti, konstanty a výčtové typy). Třída `VisualElement`, rodičovská třída většiny UI elementů, má vlastnost `Resources` typu `ResourceDictionary` (kde klíč je typu `string` a hodnota typu `objekt`). Klíč se v jazyce XAML definuje pomocí `x:Key`. Uložení hodnoty `MarginValue` do slovníku třídy `ContentPage` by vypadalo následovně:

```
<ContentPage.Resources>
    <ResourceDictionary>
        <Thickness x:Key="MarginValue">5</Thickness>
    </ResourceDictionary>
</ContentPage.Resources>
```

Získání hodnoty ze slovníku je podobné, jako získání hodnoty při použití `x:Static`. Princip vysvětluje nejdelší syntaxe.

```
<Slider>
    <Slider.Margin>
        <x:StaticResource Key="{StaticResource MarginValue}"></x:StaticResource>
    </Slider.Margin>
</Slider>
```

Výše uvedou syntaxi lze zkrátit až do následující podoby.

```
<Slider Margin="{StaticResource MarginValue}"></Slider>
```

Vlastnost `Resouce` obsahuje každá třída zděděná z třídy `VisualElement`. Proto se může stát, že element bude mít více rodičů, kteří ji definují. Může potom přistupovat k objektu ve slovníku libovolného rodiče. Pokud je přístupový klíč obsažen ve více slovnících, vybere se hodnota z nejbližšího rodiče. Vyplývá to z postupu, jakým překladač jazyka XAML hledá vlastnosti využívající `StaticResource`. Nejprve prohledá `ResurceDictionary` elementu, jemuž je vlastnost nastavována a pokud zadaný klíč neobsahuje, pokračuje v hledání v přímém rodiči. Pokud klíč nenajde ani v rodiči, pokračuje v jeho rodiči. S prvním úspěchem výsledek vrátí. `ResourceDictionary` je také možné definovat na úrovni aplikace, jelikož třída `Application` také definuje vlastnost `Resources` typu `ResourceDictionary`. Tuto kolekci překladač jazyka XAML prohledá jako poslední. Hlavní výhoda `StaticResource` spočívá v použití společně se styly, které jsou popsány v 3.7.

3.6.3 Rozšíření DynamicResource

Při přiřazení hodnoty vlastnosti pomocí `StaticResource`, je hodnota přiřazena pouze jednou. Změna kolekce `ResourceDictionary` tedy nevyvolá změnu vlastnosti, kterou nastavuje. Opak ovšem platí, pokud je `StaticResource` zaměněno za `DynamicResource`. Pokud nelze najít hodnotu s uvedeným klíčem, vznikne s užitím `StaticResource` při překladu výjimka. V případě `DynamicResource` k tomu nedojde a klíč tak lze definovat za běhu programu.

Ostatní rozšíření nejsou tak hojně využívány, proto k nim bude uvedeno pouze pár slov. `x:Type` slouží k přiřazení hodnoty typu `Type`. `x:Type` s pojí s `x:Array`, definující pole prvků daného typu. Pole typu double zobrazuje následující ukázka.

```
<x:Array Type="{x:Type x:Double}">
  <x:Double>1.2</x:Double>
  <x:Double>1.3</x:Double>
  <x:Double>1.5</x:Double>
</x:Array>
```

Poslední zástupcem je `x:Null` umožňující nastavit vlastnost na hodnotu `null`.

3.6.4 Definice vlastního rozšíření

Na základě výše uvedeného lze vytvořit vlastní rozšíření jazyka XAML. Jako ukázka poslouží třída, která umožní nastavovat hodnoty `HorizontalOptions` a `VerticalOptions` na základě typu `Boolean`. Hodnota `true` odpovídá `LayoutOptions.Start` a `false` odpovídá `LayoutOptions.End`. Jelikož jsou vlastnosti `HorizontalOptions` i `VerticalOptions` typu `LayoutOptions`, bude název třídy `LayoutOptionsExtension`. Tato třída bude implementovat rozhraní `IMarkupExtension` a bude obsahovat jednu vlastnost typu `bool` nazvanou `IsStart`. Z metody `ProvideValue` pak na základě `IsStart` vrátí vhodný objekt. Implementace třídy je následující:

```
public class LayoutOptionsExtension : IMarkupExtension
{
     public bool IsStart { get; set; }
     public object ProvideValue(IServiceProvider serviceProvider)
     {
         return IsStart ? LayoutOptions.Start : LayoutOptions.End;
     }
}
```

V jazyce XAML lze k této třídě přistoupit pomocí běžné syntaxe.

```
<Label Text="Welcome to Xamarin Forms!"
        VerticalOptions="{local:LayoutOptions IsStart = True}"
        HorizontalOptions="{local:LayoutOptions IsStart = False}">
</Label>
```

Text se zobrazí v pravé horní části obrazovky. V případě použití více vlastností, se jejich nastavování odděluje čárkami.

3.7 Styly

Styly slouží k usnadnění přiřazení stejných vlastností více elementům uživatelského rozhraní. Slouží tedy k unifikaci vzhledu elementů, zároveň zamezují duplicitě kódu jazyka XAML. K jejich definici slouží třída `Style`. Pro jejich podporu napříč visuálními elementy obsahuje třída `VisualElement` vlastnost `Style` tohoto typu. Níže budou rozebrány základní vlastnosti třídy `Style`, mimo kolekce `Behaviours` a `Triggers`, které budou rozebrány později.

Vlastnost `TargetType` obsahuje typ elementu, který styl nastavuje. Vlastní nastavení je pak uloženo v kolekci `Setters` typu `IList<Setter>`. Třída `Setter` definuje dvě vlastnosti. `Property` typu `BindableProperty`, definující nastavovanou vlastnost cílového elementu. Samotná hodnota je uvedena v druhé vlastnosti `Value` typu `object`. Mezi styly je možné vytvářet rodičovské vazby. Rodičovský styl je uveden ve vlastnosti `BasedOn`. Cílený typ rodičovského stylu (ve vlastnosti `TargetType`) musí být rodičem cíleného typu děděného stylu, případně mohou být cílené typy stejné.

Styl lze vložit do kolekce `ResourceDictionary` bez parametru `x:Key`. Klíč je v takovém případě vygenerován automaticky na základě cíleného typu. Takový styl se nazývá implicitním stylem. Nelze na něj odkazovat pomocí rozšíření `StaticResource`, jak je běžné. Pokud však existuje nějaký visuální element v oblasti platnosti `ResourceDictionary` (uchovávající implicitní styl), kterému není styl zadán explicitně a pokud jeho typ odpovídá cílenému typu implicitního stylu, je na něj implicitní styl aplikován. Na elementy děděné z cílového typu se implicitní styly neaplikují. Implicitní styly mohou mít rodičovský styl, sami však rodiči být nemohou.

Výše bylo rozebráno rozšíření `DynamicResource`, kterého lze využít ke změně stylu za běhu programu. Pokud je styl přiřazen pomocí `StaticResource` a styl se za programu změní, nebude aktualizován vzhled elementů s tímto stylem. K aktualizaci však dojde při použití `DynamicResource`. S tímto přístupem souvisí vlastnost `BaseResourceKey`, která umožní dědění stylu na základě klíče, který v době překladu není definován.

Třída `Device` definující zařízení na němž je aplikace spuštěna obsahuje třídu `Styles`, která obsahuje šest statických členů typu `Style` a šest jim příslušejícím klíčům typu `string`, které umožňují dynamickou dědičnost (při umístění do `BaseResourceKey`). Styly obsažené v této třídě jsou nazývané styly zařízení. Styly zařízení cílí na element `Label` a lze z nich odvodit styly vlastní. Názvy stylu jsou následující: `BodyStyle`, `ListItemTextStyle`, `SubtitleStyle`, `CaptionStyle`, `ListItemDetailTextStyle` a `TitleStyle`. Klíče mají název shodný s přidání slova `Key` na konec názvu. Výhodou stylů zařízení je, že při

jejich použití s rozšířením `DynamicResource` jsou schopny reagovat na změnu velikosti písma nastavenou uživatelem.

3.8 Rozměry v Xamarin.Forms

V každém prostředí je třeba určitým způsobem nastavovat velikost komponent. V dřívějších dobách bylo třeba nastavovat velikosti UI komponentů definováním počtu pixelů. Problém nastal v době rozmachu displejů s různou hustotou pixelů. Byla definována jednotka DPI (*Density per Inch*), někdy nazývané PPI (*Pixels per Inch*) udávající počet pixelů na palec. Někdy se uvádí také vzdálenost středů dvou pixelů v milimetrech. Pro příklad poslouží chytrý telefon Coolpad Modena, který má rozlišení 960 x 540 s úhlopříčkou displeje 5,5 palce. Úhlopříčka tedy odpovídá délce 1101 pixelů. Hodnota DPI je rovna 200 ($\frac{1101}{5,5}$) a pixely jsou vzdáleny 0,12688 ($\frac{5,5*25,4}{1101}$) milimetrů. Telefon Sony Xperia Z5 Premium se stejnou úhlopříčkou displeje má rozlišení 3840 x 2160. Úhlopříčka je tedy obsahuje 4405 pixelů. Hodnota DPI je 801 a pixely jsou do sebe vzdáleny 0,0317 milimetru. Objekt, který bude mít stejnou délku v pixelech, bude na druhém z uvedených telefonů 4 krát menší.

Z těchto důvodů bylo třeba navrhovat UI v abstraktních jednotkách a nechat operační systém vypočítat skutečnou délku v pixelech dle rozlišení zařízení, na kterém je aplikace spuštěna. V desktopovém světě umožnila společnost Apple programátorům pracovat v jednotkách bodů. Jeden palec odpovídá 72 bodům. Společnost Windows zavedla jednotku DIP (*Device Independent Pixel*) někdy také nazvané jako DIU (*Device Independent Unit*), kdy jeden palec odpovídá 96 DIU, tato hodnota je ovšem nastavitelná uživatelem. Ve světě mobilních zařízení je ovšem situace jiná. Tyto zařízení mají většinou větší hodnotu DPI a uživatelé je drží blíže u očí. Společnost Apple i v tomto případě používá jako jednotku body. Konverze na body je však provedena odlišně. Jeden bod může odpovídat jednomu, dvěma nebo třem pixelům. Tato konverze umožní, že velikost objektu definovaná v bodech pro iPhone4 s rozlišením 640 x 960 a konverzním poměrem dva bude stejná jako na iPhone3 s rozlišením 320 x 480 a konverzním poměrem jedna. Mobilní zařízení společnosti Apple mají navíc dynamický počet bodů na palec pohybující se kolem hodnoty 160, viz Tab. 1.

Tab. 1 Veličiny spojené s velikostí pro zařízení iPhone

Model	iPhone 2, 3	iPhone 4	iPhone 5	iPhone 6	iPhone 6 Plus
Rozlišení v pixelech [-]	320 x 480	640 x 960	640 x 960	750 x 1334	1080 x 1920
Úhlopříčka displeje [palec]	3,5	3,5	4	4,7	5,5
DPI [-]	165	330	326	326	401
Rozlišení v bodech [-]	320 x 480	320 x 480	320 x 568	375 x 667	414 x 736
Počet pixelů na bod [-]	1	2	2	2	3
Počet bodů na palec [-]	165	165	163	163	154

iPhone 6 plus má 115 % podvzorkování, jinak by hodnota rozlišení v bodech byla 360 x 640 a počet bodů na palec 134.

Android pracuje s jednotkou DPS (*Density-Independent Pixel*). U všech zařízení počítá s 160 DPS na palec. Chytré telefony Windows Phone 7 mají pixelové rozlišení 480 x 800. Pro větší zařízení byly povoleny i jiné výšky a zavedeny DIU. Kompresní poměr pro každé zařízení byl zvolen tak, aby se výsledná šířka po vydělení touto hodnotou rovnala 480. Verze 8.1 a 10 přiřazují každému mobilnímu zařízení vlastní dělící poměr, kdy výsledná šířka se nemusí rovnat 480. Poměr je stanoven tak, že hodnota DIU na palec se pohybuje přibližně ve stejných hodnotách jak je tomu u abstraktních jednotek OS Android a iPhone. Xamarin.Forms potom přiřazuje k nastaveným hodnotám tyto abstraktním jednotky jednotlivých platforem. Lze tedy počítat s přibližnou hodnotou 160 na palec, tato hodnota se ovšem bude lišit v závislosti na platformě i použitém zařízením. Pokud to není nutné, je lepší se specifikaci velikosti úplně vyhnout. V oblasti velikosti textu takovému přístupu napomáhá i vlastnost `FontSize` objektů `Label` a `Button`, kterou lze nastavit na jednu z hodnot `Default`, `Micro`, `Small`, `Medium` nebo `Large` a nechat výpočet na operačním systému.

4 Data binding a MVVM

Data binding a návrhový vzor MVVM slouží k zjednodušení interakce jazyků XAML a C# a zajišťují striktnější oddělení dat od uživatelského rozhraní. Zajistí také jednodušší testovatelnost kódu pomocí jednotkových testů. Zdrojem této kapitoly jsou [13][24][6].

4.1 Princip Data bindingu

Podporu Data bindingu v Xamarin.Forms zajišťuje třída `BindableObject`, která byla k vidění jako rodič třídy `Element`. Třída dědí přímo z třídy `System.Object`. Existuje také třída, nazvaná `BindableProperty`. Dalo by se říci, že vznikla jakási rozšířenější forma základních komponent (objekt a vlastnost). Například třída `Entry` obsahuje devět vlastností. Obsahuje však i devět statických proměnných typu `BindableProperty`, které jsou pouze pro čtení. Jejich jména sestávají názvu vlastnosti a slova `Property` (například `FontSizeProperty`). Za povšimnutí stojí fakt, že proměnné typu `BindableProperty` jsou statické, čili nezávislé na instanci. Z toho je zřejmé, že hodnoty jsou uloženy v klasických vlastnostech. `BindableObject` obsahuje několik metod pro řízení objektů typu `BindableProperty`. Jedná se o `ClearValue`, `GetValue`, `RemoveBinding`, `SetBinding` a `SetValue`. Pomocí nich nastavuje klasické vlastnosti za použití `BindableProperty`. Vlastnost `FontSize` je implementována následujícím způsobem:

```
public double FontSize
{
    set { SetValue(Entry.FontSizeProperty, value)}
    get { return (double)GetValue(Entry.FontSizeProperty)}
}
```

`BindableProperty` tedy zastřešuje přístup k vlastnosti. Umožňuje jí definovat výchozí hodnotu, uložit hodnotu, provádět operace související s validací hodnot, reagovat na změny vlastnosti, podporovat data binding a styly. Ne všechny vlastnosti všech elementů uživatelského rozhraní jsou však definovány tímto způsobem. Objekt typu `BindableProperty` lze vytvořit pomocí čtyř statických metod `Create`, `CreateReadOnly`, `CreateAttached` a `CreateAttachedReadOnly`. Metoda `Create` má celkem deset parametrů, z nichž čtyři jsou povinné. Parametr `propertyName` specifikující jméno vlastnosti, kterou `BindableProperty` zastřešuje, `returnType` specifikující její návratový typ, `declarationType` definující typ třídy, která vlastnost definuje (například `Entry`) a `defaultValue` udávající výchozí hodnotu. Nepovinnými jsou pak `defaultBindingMode` typu `BindingMode`, a pět parametrů obsahujících delegát. První z nich, specifikovaný v `coerceValue` slouží ke změně hodnoty vlastnosti na hodnotu jinou

(například do určitého rozsahu). Delegáty `propertyChanged` a `propertyChanging` jsou vyvolané po a při změně vlastnosti. `ValidateValue` slouží k určení, zda je zadávaná hodnota platná. Poslední parametr `defaultValueCreator` slouží k vytvoření výchozí hodnoty vlastnosti. Metody spojené se změnou vlastnosti nebudou volány v případě, že byla volána metoda `SetValue` s hodnotou stejnou, jakou již zastřešovaná vlastnost obsahovala.

Návratová hodnota `BindableProperty.CreateReadOnly`, je typu `BindableProperyKey`, obsahující vlastnost `BindableProperty` typu `BindableProperty`, pro kterou ovšem volání metody `SetValue` způsobí vznik výjimky (metoda se dá volat pouze v metodě `set` zastřešované vlastnosti s privátním modifikátorem). Vlastnosti `Height` a `Width` třídy `VisualElement` jsou definovány přesně tímto způsobem. Zbývá rozebrat poslední možnost vytvoření pomocí `BindableProperty.CreateAttached` a její obdoby pro vytvoření objektu pouze pro čtení `BindableProperty.CreateAttachedReadOnly`. Takto vytvořené vlastnosti umožňují nastavení vlastnosti jedné třídy pomocí instance třídy jiné. Využití nachází zejména ve spojení s potomky třídy `Layout`. Tímto způsobem je například umožněno nastavit umístění elementu v třídě `Grid` na správný řádek či sloupec (vlastnosti `Row` a `Column`). Element umístěný v mřížce vyvolá metodu `SetValue` s prvním parametrem obsahující statickou proměnnou třídy `Grid` a druhým parametrem obsahujícím hodnotu. Například `v.SetValue(Grid.Row, 2)`. Tím je hodnota vlastnosti uložena do slovníku řízeného třídou `BindableObject`. Třída `Grid` tuto hodnotu získá při umísťování elementů pomocí `GetValue` příslušného elementu. Třída `Grid` bude podrobně popsána níže, zde byla použita pouze z demonstračních účelů.

Data binding je vždy definován svým zdrojem a cílem. Zdrojem je vlastnost objektu, která se za běhu programu mění. Taková změna je následně promítnuta do vlastnosti jiného objektu, nastavené jako cíl. Cílová vlastnost musí být zastřešena `BindableProperty`, což splňuje většina vlastností visuálních elementů. Zdroj naopak musí cíl nějak informovat o změně své hodnoty. K této funkcionalitě se používá rozhraní `INotifyPropertyChanged`. Toto rozhraní obsahuje jednu událost `PropertyChanged`, vyvolanou kdykoliv dojde ke změně vlastnosti. Tato událost je typu `ProperyChangedEventHandler`. Obslužná metoda pro `PropertyChanged` má jeden parametr typu `string` specifikující název vlastnosti, která byla změněna. Aby se předešlo chybám v tomto volání, používá se často atributu `CallerMemberName`. Třída `BindableObject` implementuje rozhraní `INotifyPropertyChanged` takže elementy uživatelského rozhraní mohou být i zdrojem data bindingu. Nicméně častější je volit zdroj v třídě jazyka C#. Sestavování vazby mezi objekty sestává z dvou kroků

1. Vlastnost `BindingContext` cíle definovaná v třídě `BindableObject` je nastavena na objekt zdroje. Tím je sestavena vazba mezi objekty.
2. Pomocí metody `SetBinding` třídy `BindableObject` je sestavena vazba mezi vlastnostmi, kde první parametrem je vlastnost cíle (typu `BindableProperty`) a druhým je řetězec specifikující název vlastnosti zdroje.

Data binding na pozadí nejprve prozkoumá, zda zdroj implementuje `INotifyPropertyChanged`. Pokud ano, přidá obslužný kód události `PropertyChanged`, který způsobí přenesení hodnoty do cíle. Jednoduchým příkladem je zvětšování textu elementu `Label` (cíl) pomocí elementu `Stepper` (zdroj). V jazyce C# je implementace následující:

```
label.BindingContext = stepper;
label.SetBinding(Label.FontSizeProperty, "Value");
```

Implementace pomocí jazyka XAML vyžaduje vysvětlení dalších rozšíření, o kterých doposud nebyla řeč. `x:Reference` umožňuje odkazování na elementy uživatelského rozhraní přímo v jazyce XAML pomocí vlastnosti `Name`, což je vlastnost výchozí. `Path` u rozšíření `Binding` slouží pro nastavení zdrojové vlastnosti. Jedná se také o výchozí vlastnost.

```
<Stepper x:Name="Stepper"></Stepper>
  <Label  BindingContext="{x:Reference Name =Stepper}"
          FontSize="{Binding Path =Value}">Hello</Label>
```

Existuje i jiná varianta k sestavení vazby, než je nastavení vlastnosti `BindingContext`. V této variantě se využívá třídy `Binding` dědící z `BindableBase`. `Binding` definuje vlastnosti `Source` typu `object` pro určení zdroje a `Path` typu `string` pro určení zdrojové vlastnosti. Obsahuje také vlastnosti `Converter` a `ConverterParameter`, které slouží pro přizpůsobení hodnoty přenášené mezi zdrojem a cílem a bude o nich pojednáno později. Princip tedy spočívá ve vytvoření instance typu `Binding` s vlastnostmi `Source` a `Path` specifikující zdrojový objekt a zdrojovou vlastnost a následné přiřazení cíli pomocí metody `SetBinding`.

```
Binding b = new Binding { Source = stepper, Path = "Value" };
label.SetBinding(Label.FontSizeProperty, b);
```

Následuje ukázka této varianty v jazyce XAML.

```
<Stepper x:Name="stepper"></Stepper>
  <Label FontSize="{Binding Source={x:Reference Name =stepper},
                    Path =Value}">Hello</Label>
```

Druhá varianta je vhodná v případě, kdy má jeden objekt nastaveno více zdrojů (je ovládán více objekty). `BindingContext` lze totiž nastavit pouze na jeden objekt. Výhodou první varianty je, že `BindingContext` je propagován na podřízené elementy visuálního stromu (ty jej však mohou přepsat).

Metoda `SetBinding` byla uvedena ve variantě s druhým parametrem typu `Binding` a typu `string`. Na pozadí ovšem vytváří objekt typu `Binding`. Forma s textovým parametrem přesně odpovídá formě následující:

```
label.BindingContext = FontSizeStepper;
label.SetBinding(Label.FontSizeProperty, new Binding { Path = "Value" });
```

Výše bylo uvedeno, že cíl data bindingu je aktualizován pomocí zdroje ve spolupráci s rozhraním `INotifyPropertyChanged`. Někdy je ovšem vhodnější, aby naopak cíl ovlivňoval zdroj, případně aby byla vazba sestavena oběma směry. K tomuto účelu slouží výčtový typ `BindingMode`, který Obsahuje čtyři hodnoty. Hodnota `OneWay` značí výše popsané (změna zdroje ovlivňuje cíl), `OneWayToSource` značí situaci opačnou (změna cíle ovlivňuje zdroj). Při použití `TwoWay` je ovlivněn jak zdroj při změně cíle, tak cíl při změně zdroje. Hodnota `Default` je výchozí hodnota, předávaná metodě vytvářející `BindableProperty`.

4.2 Rozhraní ICollectionChanged

Výše bylo uvedeno, jakým způsobem lze aktualizovat element uživatelského rozhraní na základě změny v kódu na pozdí či jiného elementu uživatelského rozhraní. Problém nastává při použití kolekcí. Kolekce se typicky zobrazují pomocí elementu `ListView`. Ten má vlastnost `ItemsSource`, které je kolekce přiřazena. Typicky užívanou kolekcí v jazyce C# je generická třída `List<T>`. V případě jejího přiřazení do vlastnosti `ItemSource` funguje vše dobře do té doby, dokud není s kolekcí manipulováno ve smyslu přidání či odebrání prvků. `ListView` totiž není žádným způsobem schopen tuto změnu detekovat a aktualizovat uživatelské rozhraní. Výše zmíněné rozhraní `INotifyPropertyChanged` slouží pro informování o změně vlastností. Existuje však podobné rozhraní `INotifyCollectionChanged` pro kontrolu změny kolekcí. Podobné je v tom smyslu, že definuje jednu událost `CollectionChanged` vyvolanou při změně kolekce. Změna kolekce zde znamená přidání prvku, odebrání prvku či vyprázdnění kolekce. Událost však není vyvolána při změně prvků v kolekci. Pokud je třeba taková funkcionalita, je třeba, aby prvky kolekce implementovaly rozhraní `INotifyPropertyChanged`. Je možné vytvářet vlastní třídy implementující rozhraní `INotifyCollectionChanged`. Lze však využít i kolekci `ObservableCollection`. Tato kolekce je podobná kolekci `List<T>`, implementuje však `INotifyCollectionChanged`. Po jejím přiřazení vlastnosti `ItemsSource` bude již uživatelské rozhraní aktualizováno při změnách v kolekci.

4.3 Rozhraní IValueConverter

V praxi vznikla potřeba řídit hodnotu jistého datového typu pomocí hodnoty jiného datového typu. Typickým příkladem je změna barvy pozadí typu `Color` ovlivňovaná hodnotou zadanou do elementu `Entry` (typu `string`). Pokud uživatel zadá hodnotu správně, bude pozadí zelené, jinak červeně. K takovým účelům lze využít rozhraní `IValueConverter`. Toto rozhraní definuje metody `Convert` a `ConvertBack`. Metoda `Convert` je volána při předání hodnoty od zdroje k cíli. Metoda `ConverBack` je naopak volána při předání hodnoty od cíle ke zdroji. Obě metody mají stejné parametry. Prvním je `value` typu `object`, určující hodnotu, která by byla předána bez použití `IValueConverter`. Druhým je `targetType` typu `Type`, který udává datový typ cíle v případě metody `Convert` a datový typ zdroje v případě metody `ConvertBack`. Třetí parametr je typu `object` a umožňuje specifikovat parametr konverze. Posledním parametrem je `culture` typu `CultureInfo` udávající jazykovou verzi. Metoda `ConvertBack` nebude vyvolána v případě módu `OneWay` a obdobně metody `Convert` nebude vyvolána při použití `OneWayToSource`. V příkladu uvedeném výše bylo třeba převést hodnotu typu `string` na hodnotu typu `Color`. Dejme tomu že je od uživatele očekáváno zadání celočíselné hodnoty. Metoda `Convert` se tedy pokusí převést zadanou hodnotu na celé číslo a vrátí příslušnou barvu. Metoda `ConvertBack` jednoduše vrací `null`, jelikož nedává smysl určovat řetězec na základě barvy. Implementace je tedy následující:

```
public class BoolToBackgroundConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
          CultureInfo culture)
    {
        int i;
        return Int32.TryParse((string)value, out i) ? Color.Green : Color.Red;
    }
    public object ConvertBack(object value, Type targetType, object parameter,
          CultureInfo culture)
    {
        return null;
    }
}
```

V jazyce XAML je instance této třídy vložena do slovníku `Resources` s parametrem `x:Key` nastaveným na hodnotu `converter`. Element `Entry` využívající výše uvedenou třídu je definován následovně:

```
<Entry x:Name="Entry" BackgroundColor="{Binding Source= {x:Reference Entry},
    Path=Text, Converter={StaticResource converter}, Mode=OneWay}"></Entry>
```

4.4 Návrhový vzor MVVM

Návrhový vzor MVVM (*Model View View-Model*) je jedním z návrhových vzorů majících za úkol oddělení jistých částí kódu. Tato myšlenka byla využita již při návrhovém vzoru MVC (*Model View Controler*). MVVM z tohoto návrhového vzoru vychází. Byl však vyvinut firmou Microsoft a je tak upraven přímo na míru jazyku XAML a data bindingu.

Návrhový vzor dělí kód na tři části: model, view a view-model. View definuje uživatelské rozhraní (běžně se jedná o kód jazyka XAML). Model zastává funkci získávání dat (patří sem datové třídy s případným přístupem do databáze, souborů či dalších úložišť). View-model slouží k výměně dat mezi modelem a uživatelským rozhraním (view). Oddělení těchto částí spočívá v tom, že model by neměl být závislý na view-modelu a ten by neměl být závislý na view. View potom získává data voláním metod či přístupem k vlastnostem view-modelu. Ten stejným způsobem získává data z modelu. Pokud dojde ke změně dat v modelu, může o tom informovat view-model pomocí události. Změna view-modelu by mohla ovlivnit view stejným způsobem, preferuje se ovšem data binding. V tomto případě je view-model zdrojem a elementy uvnitř view cílem data bindingu. Z toho vyplývá, že vlastnosti ve view-model by měly implementovat `INotifyPropertyChanged`. Kód na pozadí je co možná nejvíce omezen a elementy uživatelského rozhraní v něm nejsou používány. Někdy to ovšem není zcela možné. Lze se tedy setkat i s vytvářením instance view-modelu v kódu na pozadí. Při striktním dodržení MVVM by však měl kód na pozadí obsahovat pouze volání metody `InitializeComponent`.

Uživatelské rozhraní musí být schopné reagovat na interakci uživatele (například stisk tlačítka). Výše bylo zmíněno, že k tomuto účelu lze u tlačítka využít událost `Clicked` (jiné elementy mají jiné události). Tato koncepce je ovšem v rozporu s MVVM, jelikož na tyto události je třeba reagovat v kódu na pozadí, který má obsahovat pouze volání metody `InitializeComponent`. Některé třídy (`Button`, `MenuItem`, `SearchBar`, `TextCell`, `ListView`, `TapGestureRecognizer`) obsahují vlasti `Command` typu `ICommand` a `CommandParameter` typu `object`. View-model potom obsahuje jednu vlastnost typu `ICommnad`, která je propojena data bindingem s vlastností stejného typu UI elementu. Rozhraní `ICommand` definuje dvě metody a jednu událost. První metodou je metoda `Excute` s jedním parametrem typu `object`. Tato metoda je vyvolána při interakci uživatele s UI elementem. Další metoda, `CanExecute` má parametr typu `object` a návratovou hodnotu typu `Boolean` určující, zda má uživatel možnost s UI elementem interagovat (při použití s tlačítkem je funkce návratové hodnoty stejná jako nastavení vlastnosti `IsEnabled`). S touto metodou souvisí událost `CanExecuteChanged`, vyvolaná v okamžiku změny možnosti interakce uživatele. Událost `CanExecuteChanged` lze vyvolat pomocí metody `ChangeCanExecute`. V Xamarin.Forms implementují rozhraní `ICommad` dvě třídy. Třída

`Command` a generická třída `Command<T>`. Generická verze slouží pro specifikaci parametru metody `Execute` viz dále. Konstruktory těchto tříd obsahují dva parametry. První je typu `Action` nebo `Action<object>` pro negenerickou verzi a `Action<T>` pro generickou verzi, definující akci vykonanou metodou `Execute`. Druhý parametr je volitelný typu `Func<bool>` nebo `Func<object, bool>` pro negenerickou verzi a typu `Func<T,bool>` pro generickou verzi. Takto specifikovaná funkce definuje akci vykonanou metodou `CanExecute`. Pokud není druhý parametr specifikován, vrací `CanExecute` vždy hodnotu `true`.

5 Elementy dědící z třídy Layout

Výše bylo uvedeno, že elementy dědící z třídy `Layout` mají za úkol umístit visuální elementy na obrazovku dle specifických pravidel. Jako zástupce byl rozebrán `StackLayout`. Z grafu dědičnosti bylo také patrné, že třída `Layout` má tři potomky. Konkrétně `ContentView`, `Layout<T>` a `ScrollView`. `ContentView` je rodičem pro `Frame` a `Layout<T>` je rodičem pro `AbsoluteLayout`, `Grid`, `RelativeLayout` a již zmíněný `StackLayout`. Všechny tyto elementy budou rozebrány v této kapitole.

Třída `Layout` obsahuje mnoho veřejných či chráněných metod, volaných při různých událostech spojených s umísťováním elementů na obrazovku. Tyto metody jsou klíčové pro vytváření vlastních tříd dědících z třídy `Layout` nebo jejich potomků. Dále obsahuje událost `LayoutChanged` informující o změně rozložení elementů. Třída definuje pouze dvě vlastnosti. `Padding` typu `Thickness` umožňuje nastavit odsazení elementu, kdy okraj je součástí elementu (na rozdíl od vlastnosti `Margin`). Druhou vlastností je `IsClippedToBounds` typu `Boolean` určující, zda budou elementy vložené do třídy `Layout` oříznuty při překročení jeho hranice.

5.1 ContentView a Frame

`ContentView` umožňuje zobrazit pouze jeden element. Ten je přiřazen pomocí vlastnosti `Content` typu `View`. `ContentView` se často využívá při tvorbě vlastních elementů uživatelského rozhraní, sestávajících z již existujících elementů.

`Frame` je potomek `ContentView` a slouží k zobrazení obdélníkového rámečku kolem elementu specifikovaného výše zmíněnou vlastností `Content`. Rámeček je ve výchozím nastavení odsazen, jelikož `Frame` má výchozí hodnotu vlastnosti `Padding` nastavenou na 20. `Frame` definuje vlastnosti `OutlineColor` typu `Color` udávající barvu rámečku a `HasShadow` typu `Boolean` udávající, zda má být zobrazen stín za elementem (vlastnost je funkční pouze na zařízeních s iOS).

5.2 ScrollView

Dříve bylo uvedeno, že element `ListView` umožňuje ve výchozím nastavení rolování, pro zobrazení elementů, které se nevejdou na stránku. Touto vlastností však většina elementů uživatelského rozhraní nedisponuje. Pro umožnění této funkcionality je možné využít elementu `ScrollView`. Rolování je možné horizontální, vertikální či oběma směry, dle

vlastnosti `Orientation` typu `ScrollOrientation`. Stejně jako `ContentView` obsahuje `ScrollView` vlastnost s názvem `Content` typu `View` obsahující element, jež obaluje. `ScrollView` nejprve určí, jak velký je jím obalený element. Tuto hodnotu uloží do vlastnosti `ContentSize` typu `Size`, která je pouze pro čtení. Pomocí této hodnoty a místa mu přiděleného rozhodne, zda je třeba obsah rolovat. Pokud ano, a zároveň je vlastnost `Orientation` nastavena na směr, ve kterém je vkládaný element větší než `ScrollView`, zobrazí rolovátko. Do vlastností pouze pro čtení `ScrollX` a `ScrollY` typu `Double` potom ukládá současnou hodnotu posunutí.

5.3 Layout<T> a jeho potomci

Generická třída `Layout<T>` rozšiřuje třídu `Layout` tím způsobem, že umožňuje přiřazení více elementů do jednoho rodiče. K této funkcionalitě používá kolekci `Children` typu `IList<T>`. Generický parametr typu `T` musí být třída `View` nebo její dědic.

5.3.1 StackLayout

O této třídě bylo již pojednáno výše. Umožňuje zobrazit elementy (které se umístí do vlastnosti `Children`) vedle sebe nebo pod sebou na základě vlastnosti `Orientation`. Elementy jsou oddělené mezerou, jejíž velikost udává vlastnost `Spacing`. `StackLayout` má specifický význam vzhledem k vlastnostem `HorizontalOptions` a `VerticalOptions`. Tyto vlastnosti jsou typu `LayoutOptions`, s hodnotami `Start`, `Center`, `Fill`, `End` a jejich variantami `StartAndExpand`, `CenterAndExpand`, `FillAndExpand` a `EndAndExpand`. Právě varianty končící slovem „Expand“ jsou pro `StackLayout` specifické. Pokud totiž nemá ani jeden element v kolekci `Children` nastavenu vlastnost `HorizontalOptions` či `VerticalOptions` na jednu z těchto vlastností, chová se `StackLayout` následovně. Každému elementu přidělí právě tolik místa, kolik element vyžaduje. Výška či šířka elementu `StacktLayout` je tedy v tomto případě rovna součtu výšek či šířek elementů v kolekci `Children` (v závislosti na orientaci elementu `StackLayout`). V případě nastavení `HorizontalOptions` či `VerticalOptions` na jednu z vlastností, jejíž název končí slovem „Expand“, se však `StackLayout` roztáhne na maximální možnou výšku či šířku (v závislosti na orientaci). Přebytečná výška či šířka vytváří mezery mezi elementy právě dle `HorizontalOptions` či `VerticalOptions`. Například při nastavení `StartAndExpand` je element umístěn na začátek (nahoru nebo doleva) a za ním následuje volné místo. Ostatní hodnoty výčtu `LayoutOptions` se chovají obdobně. Pokud je takto nastaveno více elementů v kolekci `Children`, je volný prostor rovnoměrně rozdělen mezi ně.

5.3.2 AbsoluteLayout

`AbsolutLayout` vyžaduje přesnou specifikaci rozměrů a umístění elementů. Vlastnosti `HorizontalOptions` a `VerticalOptions` zde nemají žádný význam. Umístění je specifikováno strukturou `Rectangle`. Tu lze specifikovat pomocí čtyř hodnot (souřadnice x, souřadnice y, šířka a výška). Druhá možnost vytvoření je pomocí struktury `Point` (definující souřadnice) a velikost struktury `Size` (definující rozměry). Využívají se zde přiřazené vlastnosti (`AttachedProperty`), o nichž bylo již dříve řečeno, že umožňují nastavení vlastnosti jedné třídy (v tomto případě `AbsoluteLayout`) pomocí instance třídy jiné (která je obsažena v kolekci `Children`). `AbsoluteLayout` definuje dvě takové vlastnosti. `LayoutBaundsProperty` pro nastavení pozice a rozměrů elementu (lze použít hodnotu `autosize` pro nastavení výchozí velikosti). Pomocí druhé, `LayoutFlagsProperty` lze zadávat rozměry či pozici relativně k velikosti elementu `AbsoluteLayout`. Toho lze dosáhnout pomocí výčtu `AbsoluteLayoutFlags` sestávajícího z osmi možností. `XProportional` a `YProportional` nastaví relativní umísťování ve směru osy X případně Y. Při použití `WidthProportional` a `HeightProportional` je zase relativně nastavována šířka nebo výška elementu. Další možnosti jsou kombinace předchozích. Nastavení `PositionProportion` odpovídá nastavení `XProportional` i `YProportional` zároveň. Podobně `SizeProportional` odpovídá nastavení `WidthProportional` i `HeightProportional`. `All` nastaví všechny pozice i velikosti relativně, naopak hodnota `None` žádnou. Pokud bude tedy například nastaveno `HeighProportional` a jako výška bude uvedena hodnota 0,5, znamená to, že element bude mít poloviční výšku oproti elementu `AbsoluteLayout`.

5.3.3 Grid

Element `Grid` umožnuje uspořádání visuálních elementů do tabulky (definované pomocí řádků a sloupců). Řádky a sloupce jsou definovány ve vlastnostech `RowDefinitions` respektive `ColumnDefinitions` typu kolekce objektů typu `RowDefinition` respektive `ColumnDefinition`. `RowDefinition` obsahuje vlastnost `Height` (pro specifikací výšky) a `ColumnDefinition` zase `Width` (pro specifikaci šířky), obě typu `GridLength`. Struktura `GridLength` obsahuje vlastnost výčtového typu `GridUnitType`. Tento výčet obsahuje hodnoty `Absolute`, `Auto` a `Star`. Použití prvního znamená, že rozměry budou zadány číselně. Druhá možnost zajistí automatické roztažení dle elementů vložených do buňky (podobně jako v případě elementu `StackLayout`). Poslední možnost se využívá k přiřazení relativní velikosti. V jazyce XAML lze před hodnotou `"*"` specifikovat číslo určující váhu. Pro příklad poslouží element `Grid` s jedním řádkem a čtyřmi sloupci. První sloupec bude mít nastavenu šířku na 200, druhý na `Auto`, třetí na hodnotu `"3*"` a čtvrtý na hodnotu `"*"`.

Rozměr prvního sloupce bude 200 jednotek, rozměr druhého právě tolik, kolik vyžadují elementy do něj vložené. Zbylé místo mřížky bude rozděleno mezi třetí a čtvrtý sloupec v poměru 3:1. Umístění prvků do mřížky potom probíhá pomocí přiřazených vlastností `Row` a `Column` s číslováním od nuly. Přiřazené vlastnosti `RowSpan` a `ColumnSpan` umožňují roztažení elementů do více řádků či sloupců. Řádky a sloupce jsou od sebe odděleny mezerou, jejíž šířka je definována vlastností `RowSpacing` pro řádky a `ColumnSpacing` pro sloupce. Typ těchto vlastností je `Integer` a výchozí hodnota je 6. Element `Grid` se hodí pro definování takzvaného gumového designu, který se dokáže přizpůsobit různým zařízením. Na základě velikosti obrazovky lze totiž jednotlivé elementy přemístit do vhodných oken mřížky (jsou jim změněny hodnoty `Row` a `Column`). Tímto způsobem lze například umístit elementy vedle sebe na větších zařízeních a pod sebou na zařízeních menších.

5.3.4 RelativeLayout

Element `RelativeLayout` umožňuje umístění elementů relativně k jiným elementům nebo sám k sobě. Funkcionality je dosaženo pomocí čtyř přiřazených vlastností `XConstraint`, `YConstraint`, `WidthConstraint` a `HeightConstraint`. `XConstraint` ovlivňuje vlastnost `X`, `YConstraint` vlastnost `Y`, `HeightConstraint` vlastnost `Height` a `WidthConstraint` vlastnost `Width`. Hodnoty jsou pro všechny tyto vlastnosti zprostředkovány pomocí rozšíření jazyka XAML definovaného ve třídě `ConstraintExpression`. Toto rozšíření obsahuje vlastnosti `Costant` a `Factor` typu `Double`, `ElementName` a `Property` typu `String` a `Type` typu `ConstraintType`. Vlastnost `Type` určuje způsob přiřazení vlastnosti. `ConstraintType` je typ výčtový a obsahuje hodnoty `Constant` pro přiřazení konstanty vlastnosti, `RelativeToParent` umožňující přiřadit hodnoty na základě hodnoty vlastnosti elementu `RelativeLayout` a `RelativeToView` přiřazující hodnoty na základě hodnoty vlastnosti jiného elementu. Pokud je použita hodnota `RelativeToView` je zdrojový element specifikován vlastností `ElementName`, jejíž hodnota se musí shodovat s názvem elementu (přiřazeným pomocí `x:Name`). Hodnota je vyčtena z té vlastnosti zdrojového elementu, jejíž název je specifikován ve vlastnosti `Property`. Výsledná hodnota je vynásobena hodnotou uloženou ve vlastnosti `Factor`.

6 Stránky

Pouze pro jednoduché aplikace postačí jedna stránka. Většinou je třeba mít stránek více a umožnit uživateli mezi nimi přepínat. Přitom je uživatelsky přívětivé mít možnost listovat historií stránek, pomocí tlačítek zpět a vpřed. Některé zařízení jsou zpětnými tlačítky vybavena (softwarovými či hardwarovými) a je třeba počítat i s nimi. Kapitola čerpá z [13][24].

6.1 Druhy stránek

Stránky lze zobrazit pomocí instancí tříd dědících ze třídy `Page`. Třída `Page` má čtyři přímé potomky. Jedná se o `TemplatedPage`, `MasterDetailPage`, `NavigationPage` a `Multipage<T>`. Potomkem třídy `TemplatedPage` je pak `ContentPage`, která byla použita ve všech doposud uvedených ukázkách. Potomky třídy `Multipage` jsou `TabbedPage` a `CarouselPage`.

6.1.1 Třída Page

Třída `Page` stojí na vrcholku hierarchie dědičnosti tříd pro zobrazení stránky. Definuje šest veřejných vlastností. `BackgroudImage` typu `string` umožnuje nastavení obrázku jako pozadí stránky. Každé stránce může být také přidělen titulek (vlastnost `Title` typu `string`) a ikona (vlastnost `Icon` typu `FileImageSource`). Stránka je oddělena od svých okrajů o vzdálenost specifikovanou vlastností `Padding` typu `Thickness`. Vlastnost `IsBusy` typu `Boolean` umožní stránce dát uživateli najevo (platformě specifickým způsobem), že je stránka zaneprázdněna a nemůže s ní interagovat. Poslední vlastností je `ToolbarItems` typu `IList<ToolbarItem>`. Pomocí této vlastnosti lze vytvořit platformě specifický panel nástrojů tvořený elementy do této kolekce vložené. Pro každý prvek panelu nástrojů lze specifikovat ikonu, jméno a prioritu. Prvky mohou být primární (lépe přístupné) nebo sekundární, což lze rozlišit vlastností `Order`. Pro obsluhu stisku lze využít vlastnosti `Command` a `CommandParameter`.

6.1.2 TempletedPage a ContentPage

Třída `TemplatedPage` definuje pouze jednu vlastnost. Jedná se o `ControlTemplate` typu `ControlTemplate`, definující vzhled stránky podobně jako `ItemTemplate` třídy `ListView`. Jediným potomkem této třídy je `ContentPage`. Tento typ stránky byl obsažen ve všech ukázkách výše a to z důvodu jednoduchosti. `ContentPage` umožňuje zobrazení jednoho elementu typu `View`, obsaženého ve výchozí vlastnosti `Content`.

6.1.3 NavigationPage

Třída `NavigationPage` umožňuje navigaci mezi stránkami. Pro podporu navigace stačí vložit instanci třídy `NavigationPage` do vlastnosti MainPage aplikační třídy. Vzhled stránky je potom mírně odlišný. Na jejím vrcholu se objeví navigační menu. Toto menu lze měnit nastavováním příslušných vlastností třídy. Titulek lze nastavit pomocí vlastnosti `Title` typu `string` definované ve třídě `Page`. Tato vlastnost neměla v případě jednostránkové aplikace význam. `NavigationPage` umožňuje nastavení barvy pozadí menu (pomocí `BarBackroudColor`) a barvu textu v menu (pomocí `BarTextColor`). Definuje také vlastnost `CurrentPage` pouze pro čtení obsahující instanci právě zobrazené stránky. Pokud je však zobrazena stránka modální, uchovává se poslední stránka běžná (rozdíl mezi modální a běžnou stránkou bude popsán níže). Vlastnost `Tint` je považována za zastaralou a neměla by se používat. Za zmínku ve spojitosti s `NavigationPage` stojí také její čtyři přiřazené vlastnosti. `HasBackButton` typu `Boolean` umožní zobrazení či nezobrazení zpětného tlačítka, `HasNavigationBar` stejného typu definuje, zda bude zobrazeno navigační menu. Zařízení s operačním systémem iOS zobrazují u zpětného tlačítka název stránky, která bude zobrazena po stisku tohoto tlačítka. Tento text lze změnit pomocí přiřazené vlastnosti `BackButtonTitle`. Poslední přiřazenou vlastností je `TitleIcon`. Zařízení Android zobrazují vedle titulku stránky ikonu. Tu lze změnit právě pomocí `TitleIcon`. Na tuto vlastnost však reagují i zařízení s iOS, které zobrazí ikonu místo titulku stránky. `NavigationPage` také definuje chráněnou metodu `OnBackButtonPressed` s návratovou hodnotou typu `Boolean`, která slouží pro obsluhu stisku hardwarového zpětného tlačítka. Přepsáním této metody lze tedy získat kontrolu nad hardwarovými zpětnými tlačítky. Tohoto přístupu se využívá například pro znemožnění uživateli návratu na předchozí stránku bez zadání potřebných údajů. O navigaci mezi stránkami bude podrobně pojednáno níže.

6.1.4 MasterDetailPage

Instance třídy `MasterDetailPage` umožňují zobrazit dvě stránky, definované ve dvou vlastnostech typu `Page`. První stránka, obsažená ve vlastnosti `Master` definuje vzhled hlavní stránky, zatímco druhá, ve vlastnosti `Detail` slouží k zobrazení stránky detailní. Mezi těmito stránkami lze přecházet, způsob přechodu je však závislý na platformě i typu zařízení. Zařízení s Windows 10 vyžadují použití objektu typu `NavigationPage` pro detailní stránku. Způsob zobrazení stránek lze nastavit pomocí vlastnosti `MasterBehavior` stejnojmenného výčtového typu. Nastavení této vlastnosti však nemá vliv na mobilní telefony (pouze na tablety a desktopové zařízení). Tento výčet obsahuje hodnoty `Default`, `Split`, `SplitOnLandscape`, `SplitOnPortraint` a `Popover`. Hodnota `Split` zapříčiní zobrazení stránek vedle sebe, kdy hlavní stránka je vlevo. `SplitOnLandscape` a `SplitOnPortrait` umožňují tuto funkcionalitu pouze při určité orientaci zařízení. Při využití `Popover` překrývá

detailní stránka částečně či úplně stránku hlavní. `Default` je potom hodnota výchozí závislá na typu zařízení a platformě. Zobrazení hlavní či detailní stránky lze řídit pomocí vlastnosti `IsPresented` typu `Boolean`. Hodnota `false` značí zobrazení detailní stránky, `true` pak zobrazení hlavní stránky. `IsGestureEnable` slouží k znemožnění uživatele přepínání mezi stránkami potáhnutím po displeji na zařízeních iOS a Android. Pro Windows 8.1 je třeba přepsat metodu `ShouldShowToolbarButton` a vrátit z ní hodnotu `false`. V každém případě však bude uživatel schopen přechodu mezi stránkami stisknutím příslušné stránky (pokud se stránky přerývají).

6.1.5 Multipage<T> a její potomci

`Multipage<T>` je generickou třídou s generickým parametrem typu `Page`. `Multipage` umožňuje uchování více stránek daného typu. Obsahuje kolekci `Children` typu `IList<T>` do které lze tyto stránky vkládat. Zobrazená stránka je pak uložena ve vlastnosti `CurrenPage`. Třída `Multipage` však definuje i vlastnosti připomínající `ListView`, který se používá pro zobrazení podobných stránek. Jedná se o kolekci `ItemsSource` typu `IEnumerable`, `SelectedItem` typu `object` odkazující na zvolený prvek z kolekce `ItemsSource` a `ItemTemplate` typu `DataTemplate` specifikující vzhled stránek v této kolekci. Potomky třídy `Multipage` jsou `TabbedPage` a `CarouselPage`.

`TabbedPage` využívá vlastností třídy `Multipage` pro umožnění přepínání mezi stránkami pomocí platformě specifického panelu (pro Android a Windows se zobrazí na vrcholu displeje, zatímco na iOS dole). V panelu jsou stránky rozlišeny titulkem (uchovávaným ve vlastnosti `Title`) a ikonou (uchovávanou ve vlastnosti `Icon`). Přidání ikony je nutné pro publikování aplikace na iOS.

Třída `CarouselPage` slouží k zobrazení stránek v podobě galerie. Xamarin.Forms však ve verzi 2.2.0 uvedl element `CarouselView`, který měl `CarouselPage` nahradit (ta by byla považována za zastaralou). `CarouselView` se však ukázal jako nestabilní a proto byl ve verzi 2.3.0 odebrán z Xamarin.Forms jeho nová verze je ke stažení v podobě NuGet balíčku. Po prokázání stability tohoto elementu bude umístěn zpět do Xamarin.Forms.

6.2 Navigace mezi stránkami

Některé z výše uvedených stránek umožňovali navigaci mezi stránkami. Existuje však i možnost jak provádět navigaci pomocí metod definovaných v Xamarin.Forms. Stránky lze dělit na dvě skupiny: modální a běžné. Modální jsou takové, které neumožní po dobu existence stránky interakci uživatele s jinou stránkou. Běžné tuto vlastnost nemají.

6.2.1 Rozhraní INavigation

Pro podporu navigace je důležité rozhraní `INavigation`. Vlastnost `Navigation` typu `INavigation` obsahuje třída `VisualElement`. `INavigation` mimo jiné definuje čtyři důležité asynchronní metody. Dvě z nich slouží navigaci na stránku. `PushModalAsync` s parametrem typu `Page`, umožňující navigaci na stránku modální a `PushAsync` se stejným parametrem, pro navigaci na stránku běžnou. Další dvě metody `PopAsync` a `PopModalAsync` slouží pro navigaci zpět a jejich návratová hodnota je `Task<Page>`. Všechny tyto metody obsahují volitelný parametr typu `Boolean` určující, zda má být přechod na stránku či ze stránky doprovázen animací. Výchozí hodnota je `true`, čili animace se provádí. Po vykonání těchto metod je bezpečné používat objekt stránky, na kterou je navigováno. `INavigation` obsahuje dvě vlastnosti typu `IReadOnlyList<Page>` pouze uchovávající historii stránek v kolekci typu zásobník. První z nich, `ModalStack`, uchovává modální stránky. Druhá, `NavigationStack`, zase běžné stránky. Přitom z běžné stránky lze přejít na stránku běžnou či modální, ze stránky modální lze přejít pouze na stránku modální. Modifikaci zásobníku `NavigationStack` lze provádět pomocí metod `PopToRootAsync`, `RemovePage` a `InsertPageBefore`. `PopToRootAsync` provede navigaci na poslední stránku v zásobníku a všechny ostatní stránky jsou z něj odstraněny. `RemovePage` odstraní ze zásobníku stránku odpovídající zadanému parametrem typu `Page`. `InsertPageBefore` vloží stránku specifikovanou prvním parametrem do zásobníku před stránku specifikovanou druhým parametrem.

6.2.2 Předávání dat mezi stránkami při navigaci

V mnohých případech je třeba předat stránce, na kterou je navigováno, nějaká data. Tohoto cíle lze dosáhnout několika způsoby. Všem navigačním metodám je předávána instance stránky. Proto je možné použít parametrický konstruktor, nastavovat vlastnosti, volat metody nové stránky či využívat události. Tento přístup ovšem není v souladu s návrhovým vzorem MVVM, jelikož obsluha konstruktoru musí být v kódu na pozadí, který má být co možná nejkratší. Někteří programátoři však argumentují tím, že navigace mezi stránkami je záležitostí uživatelského rozhraní a nemá být tedy zahrnuta ve view-modelu.

Xamarin.Forms obsahuje statickou třídu `MessagingCenter` umožňující sofistikovanější přenos zpráv. `MessagingCenter` obsahuje generické metody `Subscribe` pro registraci k příslušné zprávě daného odesilatele, `Unsubscribe` pro odregistrování příslušné zprávy daného odesilatele a `Send` pro odeslání zprávy. Každá z nich je přetížena ve verzi s jedním či dvěma generickými parametry. Pro všechny verze platí, že první generický parametr je stejného typu jako zdroj zprávy. Druhý parametr ve verzi se dvěma generickými parametry je stejného typu jako přenášená data. Verze s jedním parametrem tedy není schopna žádná data

přenášet. Jednotlivé zprávy jsou identifikovány řetězcem. Tento identifikátor je parametrem všech výše uvedených metod. Metody `Subscribe` a `Unsubscribe` mají další parametr typu `Object` definující příjemce zpráv. Dalším parametrem metody `Send` je zdroj zprávy, parametrem `Unsubscribe` zase cíl zprávy a `Subscribe` má oba parametry. Verze metody `Send` s dvěma generickými parametry obsahuje také parametr stejného datového typu jako je typ zprávy. Metoda `Subscribe` obsahuje parametr typu `Action<TSender>` (verze s jedním generickým parametrem) a `Action<TSender, TArgs>`(verze s dvěma generickými parametry). Tyto akce jsou vykonány po přijetí zprávy.

K předávání zpráv mezi stránkami lze využít i instanci aplikace (dědící z třídy `Application`). Tato instance vzniká při startu aplikace a je odstraněna při jejím ukončení a je dostupná pomocí statické vlastnosti `Application.Current`. Využití instance aplikace se však hodí spíše pro menší aplikace. Při psaní větších aplikací se metoda stává nepřehlednou. Lze využít i jiné úložiště dat například IoC (*Inversion of Control*) kontejner.

7 Platformě specifické API

Programátoři se při vývoji pro Xamarin.Forms snaží koncipovat kód takovým způsobem, aby mohl být sdílen všemi platformami. Přestože možnosti Xamarin.Forms rostou, nastávají stále situace, kdy je třeba určitou část aplikace implementovat pomocí platformě specifického API.

Výše byla popsána třída `OnPlatform` a metoda `OnPlatform`. Ty jsou dobře využitelné pro jazyk XAML (třída) či menší části kódu (většinou metoda). Existují však i jiné přístupy, které jsou rozebrány v této kapitole. Tyto přístupy se liší v závislosti na zvoleném typu projektu (SAP nebo PCL). Dále bude rozebrán princip vývoje vlastních uživatelských elementů, kdy je také třeba využívat platformě specifický kód. Kapitola čerpá z [13][24].

7.1 Platformě specifické volání v projektu typu SAP

Projekt SAP je po spuštění aplikace součástí každého z platformě specifických projektů. To umožňuje použití speciálních direktiv preprocesoru se speciálními symboly. Jedná se o symboly `__IOS__` pro iOS, `__ANDROID__` pro Android, `WINDOWS_UWP` pro universální platformu Windows, `WINDOWS_APP` pro Windows 8.1 a `WINDOWS_PHONE_APP` pro telefony s Windows 8.1. V kódu lze potom použít známé syntaxe jako `#if`, `#elif`, `#endif`. Třídy s delšími částmi takto napsaného kódu se však stávají nepřehledné.

Jinou metodou je vytvoření stejnojmenných tříd ve shodných jmenných prostorech v platformě specifických projektech. Jelikož sdílený projekt (SAP) se stane součástí každého z těchto projektů, využije vždy verzi určenou pro danou platformu.

7.2 Platformě specifické volání v projektu typu PCL

Projekt typu PCL je knihovna oddělená od platformě specifických projektů. První varianta uvedená v případě projektu typu SAP je tak nemožná. Knihovny také nemají přístup do platformě specifickým projektů, jelikož se na ně neodkazují. Tím je znemožněna i druhá varianta. Sdílená knihovna však může použít reflexi k přístupu ke všem sestavením programu, tedy i platformě specifickým projektům. Toto řešení zastřešuje třída `DependencyService`, postup implementace je následující. Ve sdílené knihovně je definováno rozhraní. V platformě specifických projektech jsou pak vytvořeny veřejné třídy implementující toto rozhraní. Tyto třídy musí také obsahovat speciální atribut `Dependency` definovaný ve třídě `DependencyAttribute`. Parametrem tohoto atributu je datový typ třídy, která bude přístupná z projektu PCL. Třída v PCL potom získá příslušnou instanci typu jím definovaného rozhraní pomocí statické generické metody `Get` třídy `DependencyService`.

Existují i další možnosti implementace volání platformě specifického API (například pomocí návrhového vzoru dependency injection). Tyto způsoby je možné realizovat pomocí externích knihoven, například knihovny prism rozebrané v 10.1.

7.3 Vlastní uživatelské elementy

Vlastní uživatelské elementy lze tvořit zcela nové nebo použít již existujících elementů. Při použití existujících elementů je většinou vytvořena třída s podporou jazyka XAML (ve Visual Studiu označen jako Forms Xaml View). Takto vytvořená třída většinou dědí z třídy `ContentView`. Je v ní dále definováno uživatelské rozhraní s použitím již existujících prvků. Pokud je to vhodné, lze vytvořit i vlastní vlastnosti typu `BindableProperty` (samozřejmě i jiné vlastnosti a metody). Instance takto vytvořené třídy jsou pak vytvořeny kdekoli je třeba. Druhý přístup spočívá ve vytvoření vlastních rendererů (custom renderer). Tento přístup bude popsán podrobněji.

7.3.1 Použití vlastních rendererů

V druhé kapitole bylo uvedeno, že Xamarin.Forms využívá renderery k dosažení platformě specifického vzhledu visuálních elementů. Existující renderery jsou definovány v sestaveních `Xamarin.Forms.Platform.iOS`, `Xamarin.Forms.Platform.Android`, `Xamarin.Forms.Platform.UWP`, `Xamarin.Forms.Platform.UAP`, `Xamarin.Forms.Platform.WinRT`, `Xamarin.Forms.Platform.WinRT.Tablet` a `Xamarin.Forms.Platform.WinRT.Phone`. Příkladem jsou třídy `ActivityIndicatorRenderer`, `ProgressBarRenderer` a další pro každý visuální element. Speciálním typem je `ViewRenderer` s generickými parametry `TView` a `TNativeView`. `TView` je na všech platformách stejného typu (`Xamarin.Forms.View`), zatímco typ `TNativeView` je platformě specifický (`UIKit.UIView` pro iOS, `Android.Views.View` pro Android a `Windows.UI.Xaml.FrameworkElement` pro Windows). Děděním z třídy `ViewRenderer` lze vytvořit zcela libovolné elementy, které budou mít definovaný vzhled na každé z platforem. Jejich tvorba však vyžaduje vytvoření vlastních tříd dědících z platformě specifické třídy typu `TNativeView` a také třídy dědící z třídy typu `TView` (`Xamarin.Forms.View`). Platformě specifické třídy musí obsahovat atribut `ExportRenderer` jehož první parametr je typu `Type` udávající typ vytvářeného elementu a druhý, také typu `Type` udává typ děděný z `ViewRenderer<TView, TNativeView>`. Pomocí tohoto atributu je pak možné na každé z platforem přiřadit vytvořenému elementu správný renderer. Platformě specifická třída (dědící z `ViewRenderer`) použije metodu `OnElementChanged`, která je vyvolána při vzniku elementu typu `TView`. Úkolem této metody je nastavit vlastnost `Control` typu

`TNativeView` definované ve třídě `ViewRenderer` pomocí metody `SetNativeControl`. Třída `ViewRenderer` definuje také vlastnost `Element` typu `TView`. Jedná se o element, který je převáděn na element platformě specifický (ve vlastnosti `Control`). Renderer může být k elementu typu `TView` přiřazen, ale i od něj odebrán. Z toho důvodu specifikuje třída předávaná metodě `OnElementCahnged` vlastnosti `NewElement` a `OldElement`. Odebrání elementu by se projevilo hodnotou `null` vlastnosti `NewValue`. V případě že vlastnost nabývá hodnoty různé od `null`, je možné přistoupit k vlastnosti `Element` (která udržuje stejný objekt jako `NewValue`) k dosažení vlastností, specifikovanými elementem typu `TView`. Změna vlastnosti přiřazeného elementu typu `TView` způsobí volání metody `OnElementPropertyChanged` třídy `ViewRenderer`. Pomocí vlastnosti `Element` a metody `OnElementPropertyChanged` lze transformovat vlastnosti třídy `TView` na vlastnosti třídy `TNativeView` (případně provádět i další akce). Jejich obsluha je provedena v platformě specifických třídách typu `TNativeView`. Tato obsluha by měla být odebrána, když je volána metoda `OnElementChanged` a vlastnost `NewValue` jejího parametru má hodnotu `null`. Po vyvolání události může být vyžadováno, aby třída typu `TNativeView` aktualizovala vlastnost třídy typu `TView`. K tomuto účelu slouží metoda `SetValueFromRenderer` definovaná v rozhraní `IElementController`, které implementuje třída `Xamarin.Forms.Element`.[13][24]

8 Transformace, animace třídy TriggerBase a Behaviour

Transformace a animace slouží ke změně vzhledu visuálních elementů. Transformace slouží ke změně pozice elementu případně jeho rozměrů určitým způsobem. Animace zase umožňují dynamickou změnu visuálního elementu.

Třídy `Style` a `VisualElement`, které tato práce popisovala, obsahovaly vlastnosti typu `IList<TriggerBase>` a `IList<Behaviour>`. Jejich funkcí je rozšířit možnosti jazyka XAML. Pomocí třídy `TriggerBase` je umožněna obsluha událostí, či změny vlastnosti v jazyce XAML. Potomci třídy `Behaviour` umožní navíc definici vlastních vlastností typu `BindableProperty`. V této kapitole budou podrobně rozebrány jejich funkce a princip. Zdrojem kapitoly jsou [13][24].

8.1 Transformace

V prostředí Xamarin.Forms se lze setkat ze třemi druhy transformací. Roztažení (scale), kdy je změněna velikost elementů, translace (translation), kdy je element posunut horizontálně či vertikálně a rotace (rotation), kdy je element pootočen podle určitého bodu či osy. Podporu transformací zajišťují vlastnosti definované ve třídě `VisualElement`, které již byly v krátkosti popsány. Jedná se o `TranslationX`, `TranslationY`, `Scale`, `Rotation`, `RotationX`, `RotationY`, `AnchorX` a `AnchorY`, všechny typu `Double`. Jejich detailnější popis následuje v této kapitole.

8.1.1 Translace elementů

K translaci elementů slouží vlastnosti `TranslationX` a `TranslationY`. Nastavení `TranslationX` na kladnou hodnotu způsobí posun elementu vpravo. Negativní hodnota pak způsobí jeho posun vlevo. Kladná hodnota `TranslationY` posouvá element dolů, záporná potom nahoru. Posunuté elementy překryjí jiné elementy, nad které jsou přesunuty. V případě přesunu elementu, který obsahuje další elementy (například `StackLayout`), jsou přesunuty i elementy v něm umístěné. Při translaci nedochází ke změně vlastností `X` a `Y` udávajících pozici elementu.

8.1.2 Roztažení elementů

Roztažení elementů lze nastavit pomocí vlastnosti `Scale`. Nastavenou hodnotou je pak vynásobena jak výška, tak šířka elementu. Při nastavení záporných hodnota dochází kromě roztažení i k rotaci elementu o 180 stupňů horizontálně i vertikálně. Poměr stran tímto způsobem měnit nelze. Podobně jako v případě translace zůstávají hodnoty `X`, `Y`, `Width` i `Height` stejné. To znamená, že okolní elementy na tuto změnu nijak nereagují. (reagovali by například na zvětšení elementu v mřížce). Ve výchozím nastavení zůstává střed elementu po roztažení ve stejném bodě. To lze změnit pomocí vlastností `AnchorX` a `AnchorY`, jimiž se nastaví střed roztažení. Střed se nastavuje relativně k velikosti elementu. Ve výchozím nastavení mají obě vlastnosti hodnotu 0.5. Nižší hodnoty znamenají posun vlevo či nahoru, naopak vyšší způsobí posun středu vpravo či dolů. Při nastavení obou vlastností na hodnotu 0 zůstává tedy levý horní roh na místě a element se roztáhne vpravo a dolů. Střed lze nastavit i mimo element a to pomocí hodnot záporných či větších než 1.

8.1.3 Rotace

Rotace umožňuje natočení elementu v jistém úhlu (zadávaného ve stupních). Kladná čísla způsobí rotaci po směru hodinových ručiček, záporná opačným směrem. S rotací souvisí vlastnosti `AnchorX` a `AnchorY` známé z roztažení elementu, zde však slouží pro definici středu rotace. Xamarin.Forms umožňuje i rotaci, která se jeví jako by byla v třírozměrném prostoru. Toho je dosaženo pomocí vlastností `RotationX` a `RotationY`. Místo rotace kolem bodu, je totiž definována rotace kolem osy. `RotationX` způsobí pocit natočení horní či spodní strany elementu k uživateli. `RotationY` způsobí pocit natočení levé či pravé strany. Vlastnost `Rotation` definuje rotaci kolem osy Z. Vlastnost `AnchorY` ovlivní `RotationX`. Pro hodnotu 0 bude rotace probíhat kolem horní části elementu, pro hodnotu 1 naopak kolem spodní části. Podobně ovlivní vlastnost `AnchorX` vlastnost `RotationY`.

8.2 Animace

Transformace, tak jak byly představeny, probíhají okamžitě. Uživatelsky zajímavé však může být postupná transformace v čase. Tohoto cíle lze dosáhnout pomocí časovačů. Xamarin.Forms však definuje animace, s jejichž pomocí je možné dosáhnout stejného cíle bez využití časovačů. Využívá k tomu tříd `ViewExtensions`, `Animation` a `AnimationExtension`. Animace nemohou být bez použití dalších technik dostupné z jazyka XAML.

8.2.1 Třída ViewExtensions

Statická třída `ViewExtenstions` obsahuje rozšiřující metody pro třídu `VisualElement`. Třída obsahuje metodu `CancelAnimation` (nejedná se o rozšiřující metodu) s parametrem typu `VisualElement`, sloužící ke zrušení animací. Ostatní metody sloužící k zahájení animace. Metod sloužících k zahájení animace je v této třídě celkem devět. Jedná se o metody `FadeTo`, `RotateTo`, `RotateXTo`, `RotateYTo`, `RelRotateTo`, `ScaleTo`, `RelScaleTo`, `TranslateTo` a `LayoutTo`. Všechny obsahují volitelný parametr typu `UInt32` udávající délku animace v milisekundách (výchozí hodnota je 250). Všechny mají také návratovou hodnotu typu `Task<bool>`, jedná se tedy o asynchronní metody. Hodnota `true` znamená, že animace byla zrušena, hodnota `false` značí dokončení animace. Prvních sedm metod má další parametr typu `Double` udávající cílovou hodnotu animované vlastnosti. Metoda `TranslateTo` má tyto parametry dva. První udává výsledný posun v horizontálním směru a druhý ve vertikálním směru. Poslední metoda `LayoutTo` je odlišná než všechny ostatní. Její parametr je typu `Rectangle` udávající novou pozici elementu. Hlavní rozdíl však spočívá ve skutečnosti, že tato animace jako jediná ovlivňuje layout a ostatní elementy. Při této animaci dochází k volání metody `Layout`, užívané k ukládání visuálních elementů. Animovanou vlastností je vlastnost `Bounds`. Ostatní animace se chovají stejně jako transformace (jejich vlastnosti `X`, `Y`, `Height` či `Width` se nemění).

Metody `RotateXTo`, `RotateYTo` a `RotateTo` slouží pro obsluhu rotace stejně jako u transformací. Metoda `FadeTo` mění průhlednost elementu. Zbývající metody `RelScaleTo` a `RelRotateTo` slouží k relativním animacím. Pokud by byla volána metoda `ScaleTo` s parametrem 2 a animace by se dokončila, nevyvolalo by opakované volání této metody s tímto parametrem žádnou změnu (vlastnost `Scale` elementu je již nastavena na hodnotu 2). Při použití `RelScaleTo`, by však vždy došlo k roztažení velikosti na dvojnásobek již zvětšeného elementu. Při volání animačních metod je zjištěna počáteční hodnota animační vlastnosti a její hodnota cílová. Při použití absolutních metod odpovídá cílová hodnota přímo specifikovanému parametru. V případě relativních metod se jedná o hodnotu počáteční přičtenou k parametru (v případě rotace) nebo násobenou s parametrem (v případě roztažení). Vlastnosti `AnchorX` a `AnchorY` mají na animace stejný vliv jako na odpovídající transformace.

Animační metody umožňují specifikovat i speciální funkci (takzvanou easing funkci), která umožní řízení rychlosti průběhu animace. Easing funkce jsou zastřešeny třídou `Easing` obsahující metodu `Ease`. Funkce musí mít tu vlastnost, že pro vstupní hodnotu 0 je její návratová hodnota 0 a pro vstupní hodnotu 1 je návratová hodnota 1. Easing funkce lze specifikovat vytvořením objektu typu `Easing`, se zadáním parametru typu `Func<double,double>`. Ve třídě `Easing` je předpřipraveno 11 statických proměnných

typu `Easing`, které lze využít. Jedná se o `Linear`, `SinIn`, `SinOut`, `SinInOuut`, `CubicIn`, `CubicOut`, `CubicInOut`, `BounceIn`, `BounceOut`, `SpringIn` a `SpringOut`. Slova „In“ a „Out“ značí, zda bude specifický efekt proveden na začátku či na konci animace (případně obojí). Kubická funkce je například typická pomalým růstem pro malé hodnoty a velkým růstem pro velké hodnoty. Při zadání `CubicIn` bude animace probíhat rychle na začátku a pomalu na konci. Při zadání `CubicOut` bude naopak animace probíhat rychle na začátku a pomaleji na konci. Jednotlivé průběhy funkcí lze najít na [25].

8.2.2 Třída Animation

Výše uvedené přístupy vytvářejí objekty typu `Animation`. Samotná animace je spuštěna voláním jejich metody `Commit`. Třída `Animation` obsahuje bezparametrický konstruktor a konstruktor s pěti parametry. Prvním parametrem je `callback` typu `Action<double>`, který slouží pro modifikaci hodnoty animované vlastnosti. Počáteční hodnota je nastavena parametrem `start` typu `Double` s výchozí hodnotou 0. Koncová potom v parametru `end` stejného typu s výchozí hodnotou 1. Předposledním parametrem je `easing` typu `Easing`, specifikující funkci průběhu animace. Poslední parametr je `finished` typu `Action`, definující akci vyvolanou po dokončení animace. Samotná animace je započata voláním metody `Commit`. Tato metoda má sedm parametrů. Název zadaný parametrem `name` typu `string`, `rate` typu `Uint` určující periodu volání metody předané třídě `Animation` (slouží většinou pro aktualizaci uživatelského rozhraní), `length` typu `Uint` specifikující délku animace, `easing` typu `Easing` specifikující funkci průběhu animace, `repeat` typu `Func<bool>` definující funkci určující, zda se má animace opakovat, `finished` typu `Action<double, bool>` a `owner` typu `IAnimatable`. Parametr `finished` obsahuje akci volanou při dokončení animace. Prvním parametrem je finální hodnota a druhý určuje, zda byla animace přerušena (v případě hodnoty `true`). Rozhraní `IAnimatable` definuje dvě metody `BatchBegin` a `BatchCommit`. Toto rozhraní implementuje třída `VisualElement`.

Třída `Animation` implementuje rozhraní `IEnumerable` a může tedy uchovávat kolekci animací (takzvaných dceřiných animací). Instance třídy `Animation` obsahující dceřiné animace se pak nazývá rodičovskou animací. Dceřiné animace lze přidávat pomocí metod `Add`, `Insert` nebo `WithConcurrent`. Rozdíl mezi `Add` a `Insert` je, že návratovou hodnotou `Insert` je rodičovská animace, zatímco metoda `Add` má návratový typ `void`. Kromě instance tříd `Animation` jsou parametrem těchto metod i dvě hodnoty udávající počáteční a koncový čas animace relativně k době trvání rodičovské animace. Metoda `WithConcurent` umožňuje zadávání času v jiném rozmezí než 0 až 1. Hodnoty mimo tento rozsah však nejsou brány v potaz.

8.2.3 Třída AnimationExtensions

Volání metody `Commit` třídy `Animation` způsobí volání metody `Animate` třídy `AnimationExtensions`. Třída definuje mnoho metod, které rozšiřují rozhraní `IAnimatable`. Metoda `Animate` je definována ve třech variantách. Nejobecnější z nich obsahuje všechny parametry zmíněné u metody `Commit`. Existuje i generická varianta `Animate<T>` mající další parametr `transform` typu `Func<double, T>`. Tento parametr slouží k převodu typu `Double` na typ generický. Generický parametr pak předán metodě `callback`, která je v tom případě také generická. Metoda `AnimationIsRunning` s parametrem typu `string` a návratovým typem `Boolean` udává, za probíhá animace. Metoda `AboartAnimation` se stejným parametrem animaci zastaví.

8.3 Třída TriggerBase a její potomci

Třída `TriggerBase` je abstraktní třída. Funkcí jejich potomků je reakce na nějakou událost či změnu vlastnosti. Reakci provedou na základě zadané podmínky. Reakce provedené v případě změny této podmínky na pravdivou jsou uchovány ve vlastnosti `EnterActions` typu `IList<TriggerAction>`. Při změně podmínky na nepravdivou jsou vyvolány akce z `ExitActions` stejného typu. Třída `TriggerBase` obsahuje také vlastnost `TargetType` typu `Type` udávající typ elementu, ke kterému je přiřazena. Z třídy `TriggerBase` dědí třídy `Trigger`, `EventTrigger`, `DataTrigger` a `Multitrigger`.

8.3.1 Třída Trigger

Třída `Trigger` slouží k vyvolání akce na základě změny vlastnosti. K tomuto účelu využívá vlastnosti `Property` typu `BindableProperty`, `Value` typu `Object` a `Setters` typu `IList<Setter>`. Třída `Setter` byla již rozebrána v souvislosti se styly. Třída `Trigger` je tedy vhodná pro nastavování vlastností. Vlastnosti `Property` a `Value` specifikují podmínku. Ta je splněna, nabyde-li sledovaná vlastnost (definovaná vlastností `Property`) příslušnou hodnotu (definovanou vlastností `Value`). Změna pravdivostní hodnoty této podmínky na hodnotu `true` způsobí vyvolání akcí specifikovaných v `Setters`. Jsou také vyvolány události specifikované vlastnostmi `EnterActions` třídy `TriggerBase`. Změna na hodnotu `false` způsobí inverzní operaci k operaci aplikované v kolekci `Setters` a také vyvolání událostí v `ExitActions` třídy `TriggerBase`.

8.3.2 Třída EventTrigger

Třída `EventTrigger` definuje vlastnosti `Event` typu `string` a `Actions` typu `IList<TriggerAction>`. Vlastnost `Event` specifikuje událost, při jejímž vyvolání jsou

provedeny akce v kolekci `Actions`. Třída `TriggerAction` je rodičovskou třídou pro generickou třídu `TriggerAction<T>`. Obě tyto třídy jsou abstraktní a obsahují metodu `Invoke`, která bude vyvolána při vyvolání události. Generický typ pak určuje typ elementu, pro který je `EventTrigger` určen. `EventTrigger` nevyvolává akce `ExitActions` ani `EnterActions`.

8.3.3 Třída DataTrigger

Obě výše zmíněné třídy mohly manipulovat pouze s elementy, k nimž byli přiřazeny. Třída `DataTrigger` umožňuje reagovat na změnu vlastnosti objektu, jemuž není přiřazena. `DataTrigger` se velice podobá třídě `Trigger`. Má také dvě shodné vlastnosti `Value` a `Setters`. Místo vlastnosti `Property` má však vlastnost `Binding` typu `BindingBase`, umožňující specifikaci sledované vlastnosti.

8.3.4 Třída Multitrigger

Třída `Multitrigger` umožňuje vyvolání akce na základě více podmínek. Tyto podmínky spojeny logickým operátorem AND. Třída obsahuje vlastnost `Conditions` typu `IList<Condition>` a také vlastnost `Setters`. Třída `Condition` je abstraktní a má dva dědice. První je třída `PropertyCondition` s vlastnostmi `Property` a `Value`, jako třída `Trigger`. Druhou pak `BindingCondition` s vlastnostmi `Binding` a `Value` jako `DataTrigger`. Pokud jsou všechny podmínky splněny, jsou aplikovány nastavení v kolekci `Setters`.

8.4 Třída Behaviour a její potomci

Veškeré funkcionality, kterou zajišťují potomci třídy `TriggerBase`, lze dosáhnout i pomocí třídy `Behaviour` a jejich potomků. Tento přístup by byl o něco náročnější. Dědicem třídy `Behaviour` je generická abstraktní třída `Behaviour<T>`. Generický parametr je opět nejobecnějšího typu, ke kterému může být instance `Behaviour<T>` přiřazena. Nejdůležitější roli třídy `Behaviour<T>` mají metody `OnAttachedTo` a `OnDetachingFrom`, obě přetížené s parametrem generického typu a s parametrem typu `BindableObject`. Metoda `OnAttachedTo` je volána v době přiřazení instance třídy `Behaviour<T>` cílovému elementu, zatímco `OnDetachingFrom` je volána při jeho odebrání. V metody `OnAttachedTo` se tedy typicky přiřazují obsluhy událostí, které se naopak v metodě `OnDetachingFrom` odebírají. Zásadním rozdílem mezi třídou `Behaviour` a `TriggerBase` je, že třída `Behaivour` dědí z `BindableObject` a může tedy definovat vlastnosti typu `BindableProperty`. Tím se třída stává více flexibilní. Zároveň se však stává stavovou

a nemá tedy smysl ji definovat pomocí stylu. Na změnu vlastností definovaných některým z dědiců třídy `Behaviour` lze reagovat pomocí dědiců `TriggerBase`.

Pomocí výše popsané funkcionality lze tedy omezit kód na pozadí a převést jej do oddělených tříd. Tento princip je v souladu s návrhovým vzorem MVVM. Existují dokonce i přístupy, které pomocí třídy `Behaviour<T>` umožňují obsluhu události visuálního elementu pomocí vlastnosti typu `ICommand`.

9 Popis jednotek PT41 a příslušenství

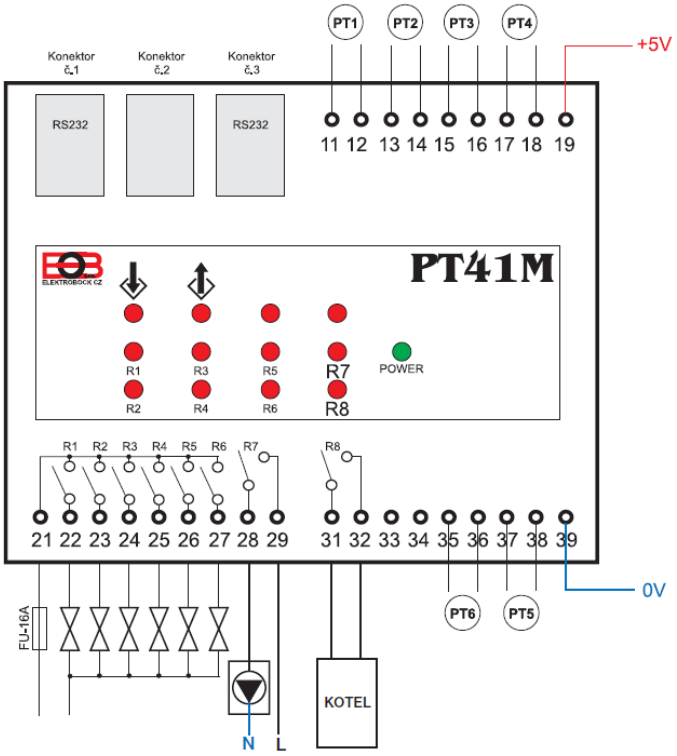
V rámci diplomové práce byla implementována aplikace sloužící k ovládání jednotek PT41 firmy Elektrobock. V této kapitole bude rozebráno schéma, funkce a možnosti připojení těchto jednotek. Kapitola čerpá z [9].

9.1 Popis jednotek PT41

Firma Elektrobock nabízí dvě varianty jednotek typu PT41. Jedná se o jednotky PT41-M (Master) a PT41-S (Slave). Jednotka PT41-M je osazena šesti vstupy s označením PT1 až PT6, na které je možné připojit teplotní čidla CT05 či prostorové termostaty. Každému vstupu odpovídá jeden reléový výstup R1 až R6. Na tyto výstupy jsou pak připojeny termostatické ventily, které ovládají topení v jednotlivých oblastech (nazývaných jako zóny). Jednotka je dále osazena dvěma dalšími reléovými výstupy s označením R7 a R8, na něž se připojuje čerpadlo (R7) a kotel (R8). Jednotky PT41 na základě měření stavu jednotlivých vstupů spínají příslušné reléové výstupy. Jednotka je osazena LED (*Light-Emitting Diode*) diodami, indikujícími její aktuální stav. Jedná se o tři řady červených diod a jednu zelenou napravo od nich. První dioda v první řadě udává aktivitu nadřazené jednotky, druhá potom aktivitu podřízení jednotky. Třetí dioda je nevyužita a čtvrtá indikuje blikáním správný chod jednotky. Druhá a třetí řada diod indikuje stav reléových výstupů, jejichž označení je vždy uvedeno pod příslušnou diodou. Svítící dioda signalizuje sepnutý výstup, zhasnutá pak rozepnutý výstup. Posledním možným stavem je blikání diody indikující nepřipojení příslušného vstupu (pouze pro R1 až R6). Poslední nepopsanou diodou je zelená dioda s označením power indikující připojení napájecího napětí. Schéma jednotky PT41-M lze vidět na Obr. 4.

Jednotka PT41-S se liší tím, že počet jejich vstupů je osm PT1 až PT8 . Vstupům PT7 a PT8 jsou přiřazeny výstupy R7 a R8. PT41-S tedy neslouží k ovládání kotle a čerpadla a slouží k rozšíření systému.

Jednotky lze zapojovat za sebe pomocí sériové linky RS232. Každá jednotka je osazena třemi porty RS232. Jednotka propojená s prvním portem sériového rozhraní je aktuální jednotce nadřazená a jednotka zapojená do třetího portu je jednotce podřízená. Druhý port je nevyužit. Maximálně lze za sebe zapojit čtyři jednotky. Jejich typ lze libovolně kombinovat. Doporučeným zapojením je konfigurace s jednou jednotkou PT41-M a maximálně třemi jednotkami PT41-S. V této konfiguraci lze připojit až 30 vstupů s příslušnými výstupy a ovládat kotel a čerpadlo.



Obr. 4 Schéma jednotky PT41-M

9.2 Funkce jednotek PT41

Základním účelem jednotek PT41 je spínání výstupů R1 až R8 na základě stavu připojených vstupů. Je-li vstupem prostorový termostat, nelze na tento vstup aplikovat další logiku, protože stav vstupu přímo určuje sepnutí/rozepnutí výstupního relé. Jiná situace nastává při připojení teplotního čidla. Jednotka kontroluje teplotu detekovanou čidlem a na základě této teploty a uživatelem nastavené požadované teploty a dalších uživatelských nastaveních spíná příslušné výstupy. Nastavení lze rozdělit do dvou skupin. Jedná se o nastavení celé jednotky, které je aplikované na všechny vstupy (dále označeno jako globální) a nastavení specifické pro daný vstup.

9.2.1 Globální nastavení

V globální nastavení lze nastavit hodnoty platné pro všechny jednotky. Některá nastavení jsou přístupná vždy, jiná pouze po zadání servisního hesla. Prvním nastavením, k němuž není třeba servisního hesla, je pracovní mód. Tento mód může být letní nebo normální. V normálním režimu jsou příslušné výstupy spínány dle požadované teploty. V letním režimu jsou všechny výstupy kromě výstupů s kotlem a čerpadlem trvale sepnuty, aby nedocházelo k opotřebení těsnění připojených ventilů vlivem jejich dlouhodobého utažení. Další popis předpokládá zvolení normálního režimu.

Dalším nastavením je útlumová teplota. Pokud je útlumová teplota využívána, nahrazuje její hodnota požadovanou teplotu v libovolném režimu. Dále lze také nastavit minimální a maximální nastavitelnou teplotu pro všechny jednotky, kdy minimální hodnotu lze nastavit v rozsahu 2 až 10 °C a maximální v rozsahu 15 až 99,5 °C. Po nastavení těchto konstant již není možné nastavit žádnému vstupu požadovanou teplotu, která by neležela v rozsahu mimo nastavené minimum a maximum.

Další využitelnou vlastností je možnost nastavení korekce teploty. Korekce teploty upravuje teplotu změřenou čidlem. K chybnému měření může docházet vlivem přídavného elektrického odporu kabelů vedoucích mezi jednotkou a čidlem (čidla jsou odporová).

K sepnutí čerpadla a kotle nemusí docházet ihned po sepnutí některého z reléových výstupů. Časový rozestup těchto událostí lze nastavit v rozmezí 0 až 5 minut.

Jednotkám PT41-M je možné nastavit, aby spínaly čerpadlo a kotel pouze na základě požadavku svých vstupů. V tomto případě jsou požadavky vstupů podřízených jednotek ignorovány.

Jednotky mohou pracovat ve dvou režimech regulace, z nichž si může uživatel zvolit pouze jeden. Účelem regulací je úspora energie a snížení opotřebení vytápěcích zařízení (kotle či čerpadla). Regulace je definována globálně, aplikována je však na každý vstup a výstup zvlášť.

Prvním typem regulace je hystereze. V režimu hystereze lze nastavit její hodnotu ve stupních a minimální čas, po který je možné výstup sepnout. Výstupní relé daného vstupu je sepnuto tehdy, je-li rozdíl požadované a změřené teploty větší než nastavená hodnota hystereze. K rozpojení výstupního relé dojde po dosažení požadované teploty, přičemž musí být dodržen minimální čas sepnutí výstupu.

Druhým typem regulace je PI (*Proportional Integral*) regulace, která nabízí jistá vylepšení. S tímto typem regulace souvisí tři hodnoty. První z nich je interval regulace v minutách, druhým je šířka pásma a třetím minimální doba sepnutí. Pokud je rozdíl měřené a požadované teploty větší než šířka pásma, je výstup sepnut stále. V opačném případě dochází s periodou určenou délkou intervalu k výpočtu dvou hodnot. První z nich určuje dobu, po kterou bude výstup sepnut a druhý dobu, po kterou bude rozepnut. Součtem těchto hodnot je nastavený interval. Výpočet intervalů je závislý na předchozích výpočtech a jedná se tedy o učící se algoritmus. Minimální doba sepnutí má stejný význam jako v případě hystereze.

9.2.2 Nastavení specifické pro každý vstup

Konstanty z této skupiny lze nastavit každému vstupu zvlášť. I v tomto případě jsou některá nastavení přístupná až po zadání správného servisního hesla, jiná i bez něj.

Servisní heslo je třeba nastavit pro definici typu připojeného zařízení k danému vstupu (prostorový termostat nebo teplotní čidlo) a pro nastavení priority daného vstupu. Pokud je vstup nastaven jako prioritní, je na sepnutí jeho výstupu reagováno sepnutím výstupů určených pro čerpadlo a kotel. V opačném případě k sepnutí nedojde. Neprioritní zóny se využívají v místech, kde není nutná okamžitá reakce na snížení či zvýšení teploty.

Ostatní nastavení jsou dostupná bez nastavení servisního hesla. Do této skupiny patří nastavení vypnutí výstupu a volba režimu.

Při nastavení výstupu jako vypnutého dojde k jeho sepnutí pouze v případě, že změřená teplota poklesne pod 3 °C.

Režim souvisí s požadovanou teplotou. Požadovanou teplotu lze vždy nastavit s přesností 0,5 °C. Vstup může být nastaven v režimu auto (automatický) nebo manu (manuální).

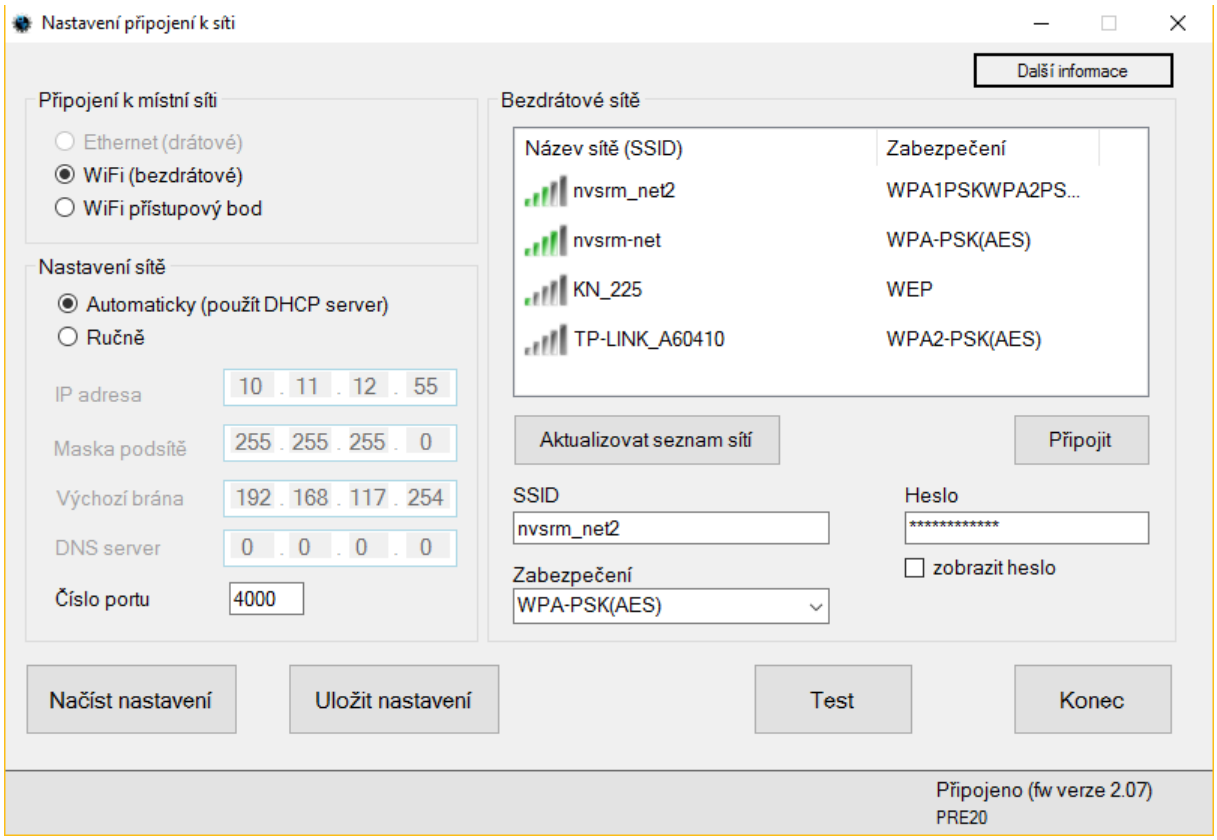
V režimu manu je požadovaná teplota konstantní až do její další změny uživatelem. Režim auto je spojen s programem, který lze nastavit každému vstupu zvlášť. Program sestává z šesti dvojic hodnot čas, teplota, kdy první hodnota udává čas od kterého je požadovanou teplotou příslušná teplota. Tuto šestici lze nastavit pro každý den v týdnu. Čas lze nastavit s přesností 10 minut. V režimu auto lze nastavit současnou požadovanou teplotu ručně. Tato hodnota bude platná do doby, kdy nastane změna v programu. Od verze 12.04 lze také nastavit používání útlumové teploty na úrovni vstupu. Po tomto nastavení bude tedy požadované teplota vstupu řízena pouze podle útlumové teploty (tu lze nastavit pouze globálně).

Obr. 5 Jednotky PT41-M(vlevo) a PT41-S(vpravo)

9.3 Možnosti komunikace s jednotkami

Komunikace s jednotkami probíhá dvěma možnými způsoby. Prvním z nich je použití standardu 802.11. V tomto případě je nutné komunikovat prostřednictvím převodníku PRE30

ETH/WIFI. Jedná se o zařízení obsahující 802.11 modul, porty pro ethernet, RS232 a USB (*Universal Serial Bus*). Pomocí zařízení lze vyhledat bezdrátové sítě, ke kterým je následně možné se následně připojit. Připojení k lokální síti je možné pomocí statické IP (*Internet Protocol*) adresy i protokolu DHCP (*Dynamic Host Configuration Protocol*). Ke konfiguraci slouží software Wifi_Ethernet_software z [9] jehož vzhled je vidět na Obr. 6.



Obr. 6 Grafické rozhraní programu Wifi_Ethernet_software

Druhou možností je připojení zařízení pomocí USB portu. V tomto případě je třeba použít převodník PRE USB/RS232. Jeho úkolem je převod dat z USB na RS232.

Ke konfiguraci jednotek dochází zapojením některého z převodníků do prvního portu té jednotky, která žádnou nadřízenou jednotku nemá. Oba převodníky lze pozorovat na Obr. 7.

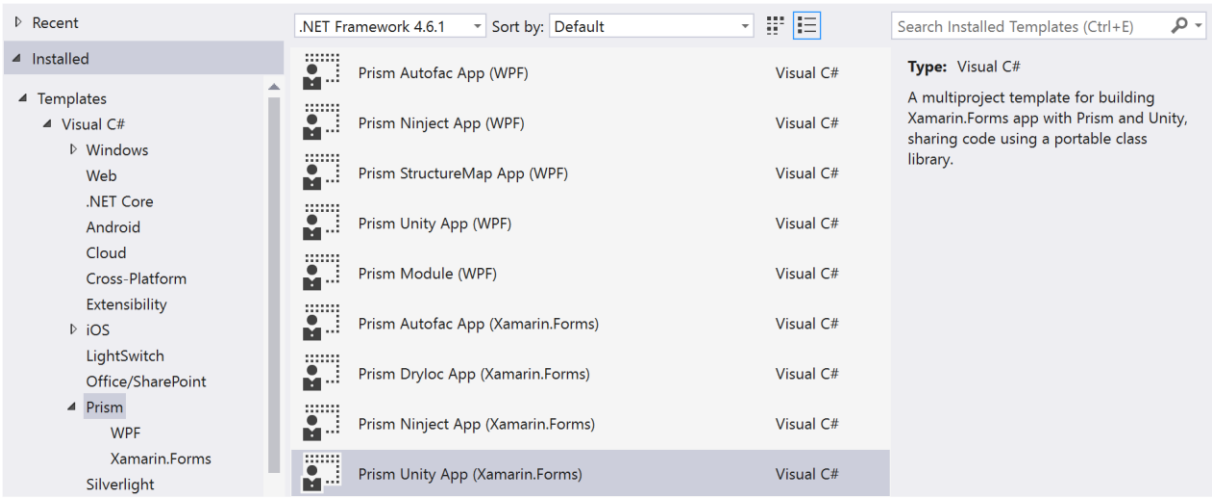
Obr. 7 Převodníky PRE30 ETH/WIFI (vlevo) a PRE USB/RS232 (vpravo)

10 NuGet balíčky použité při implementaci

Při vytváření aplikace byly využity NuGet balíčky pro dosažení přídavné funkcionality. Jedná se o knihovny Prism.Unity.Forms, sqlite-net, rda.SocketsForPCL, Acr.UserDialogs a SkiaSharp.Forms se všemi jejich závislostmi. V této kapitole budou tyto knihovny rozebrány.

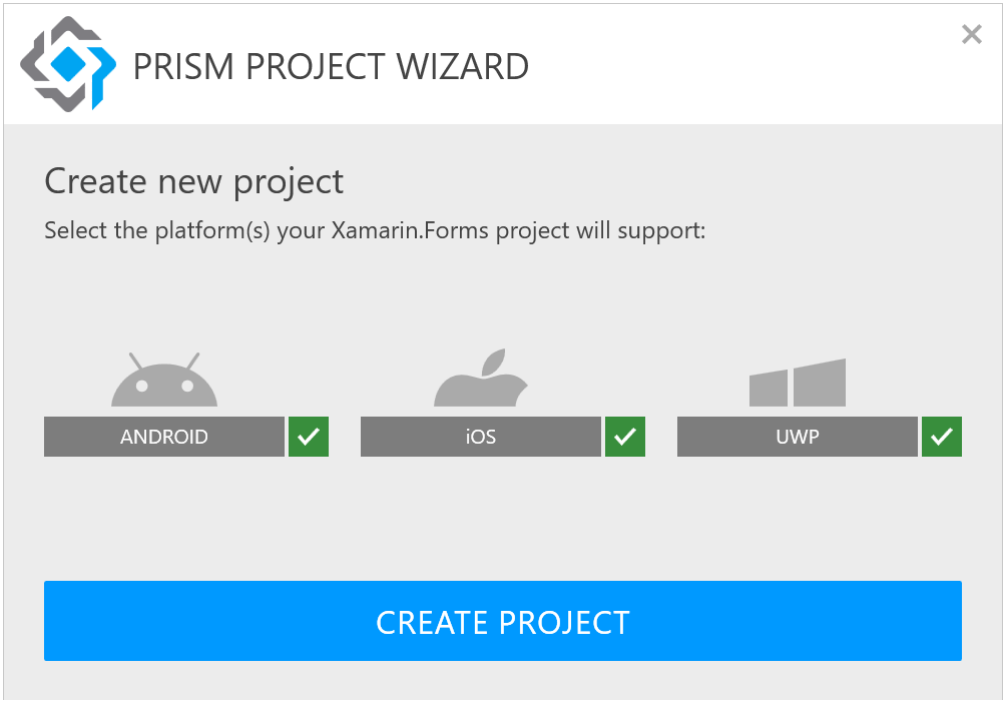
10.1 Prism.Unity.Forms

Knihovna Prism.Unity.Forms usnadňuje vývoj aplikací založených na návrhovém vzoru MVVM ve spojení s návrhovým vzorem dependency injection a za využití knihovny Unity. Autorem knihovny Prism jsou Brian Lagunas a Brian Noyes. Autorem knihovny Unity je společnost Microsoft. Tyto knihovny jsou používány nejen v Xamarin.Forms aplikacích ale také v aplikacích WPF. Prism je využitelný i ve spojení s jinými knihovnami než Unity. Tyto knihovny musí podporovat návrhový vzor dependency injection. Příkladem jsou Autofac, Ninject a další. Prism definuje vlastní šablony projektů, viz Obr. 8. Po vytvoření projektů pomocí šablony jsou nainstalovány příslušné balíčky, je vytvořena složka Views a ViewModels v projektu PCL a jsou příslušným způsobem upraveny třídy aplikace a stránek v platformě specifických projektech. V podkapitole je čerpáno z [11].



Obr. 8 Šablony projektů knihovny Prism

Šablony projektů společně se šablonami tříd a kódu popsaných níže jsou ke stažení na [14]. Verze šablon 1.1 podporuje vývoj pouze pro Android, iOS a UWP. Výběr podporovaných platforem následuje po potvrzení typu projektu v šabloně, viz Obr. 9.



Obr. 9 Dostupné platformy Prism 1.1

Třída aplikace při vytvoření projektu pomocí šablony dědí ze třídy `PrismApplication`. Třída `PrismApplication` obsahuje dvě abstraktní metody `RegisterTyper` a `OnInitialized` a konstruktor s parametrem typu `IPlatformInitializer`. Metoda `OnInitialized` slouží k volání metody `InitializeComponent` a navigaci na hlavní stránku. Navigace je obalena funkcionalitou Prism a její syntaxe je jiná než v Xamarin.Forms. K navigaci slouží instance typu `INavigationService`. Samotná navigace probíhá pomocí řetězců nazvaných uri. Uri může sestávat pouze z názvu cílové stránky, lze však s jeho pomocí tvořit i hierarchickou navigaci a předávat parametry. Hierarchické navigace lze využít zejména ve spolupráci se stránkami typu `NavigationPage`, `TabbedPage` a `MasterDetailPage`. Všechny stránky zmíněné v řetězci musí být registrovány v dependency kontejneru. Stránky v hierarchii jsou odděleny lomítkem a parametry jsou předávány za otazníkem. Tyto parametry mohou být předány v textové formě a jejich formát je `<název> = <hodnota>`. Jednotlivé parametry jsou odděleny ampersandem. Jako příklad uri poslouží následující kód.

```
NavigationService.NavigateAsync("PrismNavigationPage
/PrismContentPage1?test=ahoj/MainPage?title=Hello ");
```

Kde `PrismNavigationPage`, `PrismContnetPage1` i `MainPage` jsou existující stránky vytvořené ve složce Views. Po vykonání této metody dojde k postupné navigaci mezi stránkami, které budou přidávány do navigačního zásobníku. Stránce `PrismContentPage1` bude navíc předán parametr test s hodnotou ahoj a stránce `MainPage` parametr `title` s hodnotou Hello. Prism obsahuje logiku pro řízení aplikačního zásobníku a je možné

používat jak softwarová, tak hardwarová zpětná tlačítka k navigaci zpět. V ukázce by při stisku zpětného tlačítka došlo k navigaci na stránku `PrismContentPage1`. Metoda `NavigateAsync` obsahuje další volitelné parametry. Prvním z nich je slovník `NavigationParameters`, který lze využít pro předávání parametrů. Do tohoto slovníku jsou předávány i parametry v případě navigaci s využitím uri. Další dva parametry metody `NavigateAsync` jsou typu `bool` a určují, zda je navigováno na modální stránku a zda má být navigace animována.

Důležitou funkcionalitou je řízení životního cyklu stránky v jejím view modelu. Každý view model může implementovat rozhraní `INavigationAware`. Toto rozhraní obsahuje metody `OnNavigatedTo` a `OnNavigatedFrom` s parametrem typu `NavigationParameters` obsahujícím parametry předávané příslušné stránce.

Prism definuje i vlastní šablony sloužící pro tvorbu stránek. Po využití šablony je vytvořena stránka i view model pojmenovaný dle jmenných konvencí knihovny prism. Do metody `RegisterTypes` je vložen řádek pro registraci nově vytvořené stránky. Vytvořit lze stránky typu `CarouselPage`, `ContentPage`, `MasterDetailPage`, `NavigationPage` a `TabbedPage`.

Metoda `RegisterTypes` slouží k registraci typů či tříd do dependency injection kontejneru, který je dostupný pomocí vlastnosti `Container`. Registrovat typ je možné pomocí metody `RegisterType` a instanci pomocí `RegisterInstance`. Speciální funkcionalitu nabízí generická metoda `RegisterTypeForNavigation`. Generickým typem je třída stránky (typu `Page`). Prism po takové registraci přiřadí stránce view model tím způsobem, že se pokusí vytvořit instanci třídy umístěné ve složce ViewModels, s názvem odpovídajícím názvu stránky s přidaným textem „VievModel“ na konec. Pokud je tedy registrována stránka `SettingsPage` pomocí volání `Container.RegisterTypeForNavigation<SettingsPage>()`, je příslušný view model hledán ve složce ViewModels pod názvem `SettingsPageViewModel`. Pokud není vhodné či možné tento explicitní způsob vytváření view modelu využít, je možné pro registraci použít přetížení metody `RegisterTypeForNavigation` se dvěma generickými parametry, kdy druhým je zvolený view model. V obou případech je při vytvoření stránky do vlastnosti `BindingContext` vložena instance příslušného view modelu. Ve starších verzích knihovny Prism (nižších než 6.2.0) je pro zajištění této funkcionality nutné doplnit do konfigurace každé stránky následující řádky.

```
xmlns:prism="clr-namespace:Prism.Mvvm;assembly=Prism.Forms"
prism:ViewModelLocator.AutowireViewModel="True"
```

Parametry konstruktorů view modelů jsou automaticky naplněny daty z dependency kontejneru, pokud jsou v něm příslušné typy registrovány. Pro tvorbu view modelů je také

možné použít třídu `BindableBase` z knihovny prism implementující rozhraní `INotifyPropertyChanged`.

Od verze 6.3 obsahuje knihovna prism ve jmenném prostoru `Prism.Behaviours` třídu `EventToCommandBehaviour`. Tato třída slouží k využití návrhového vzoru MVVM pro obsluhu událostí. Třída `EventToCommadBehaviour` má parametry `EventName`, `Command`, `CommandParameter`, `EventArgConverter`, `EventArgsConverterParameter` a `EventArgsParameterPath`. `EventName` definuje název události. Vlastnosti `Command` je přiřazen objekt typu `ICommand` jehož metoda `Execute` je volána po vyvolání události s parametrem specifikovaným pomocí `CommandParameter`. Pokud je třeba předat parametr z objektu `EventArgs` události, lze využít dva přístupy. Prvním z nich je využití `EventArgsConverter` a `EventArgConverterParametr` pro extrakci parametru z objektu `EventArgs`. Druhým je definice cesty k vlastnosti pomocí `EventArgsParameterPath`, podobné parametru `Path` u rozšíření `Binding`. Následující příklad způsobí volání příkazu `GoToZonePageCommand`. Jeho parametrem bude vlastnost `Item` parametru, který je typu `ItemTappedEventArgs`.

```
    <behaviors:EventToCommandBehavior EventName="ItemTapped" Command="{Binding GoT
oZonePageCommand}" EventArgsParameterPath="Item" ></behaviors:EventToCommandBehavi
or>
```

Prism nabízí mnoho dalších možností například pro práci s platformě specifickým kódem, vlastním typem `RellayCommand` děděným z `Commad` či zobrazení platformě specifických dialogů. Pro zobrazování dialogů slouží rozhraní `IPageDialogService`. Toto rozhraní je však omezeno na dialog, v němž si uživatel může vybrat z několika možností a upozornění s přídavným jedním či dvěma tlačítky. Jelikož aplikace PT41 vyžaduje širší funkcionalitu v oblasti dialogů, je využita knihovna Acr.UserDialogs popsaná v následující podkapitole.

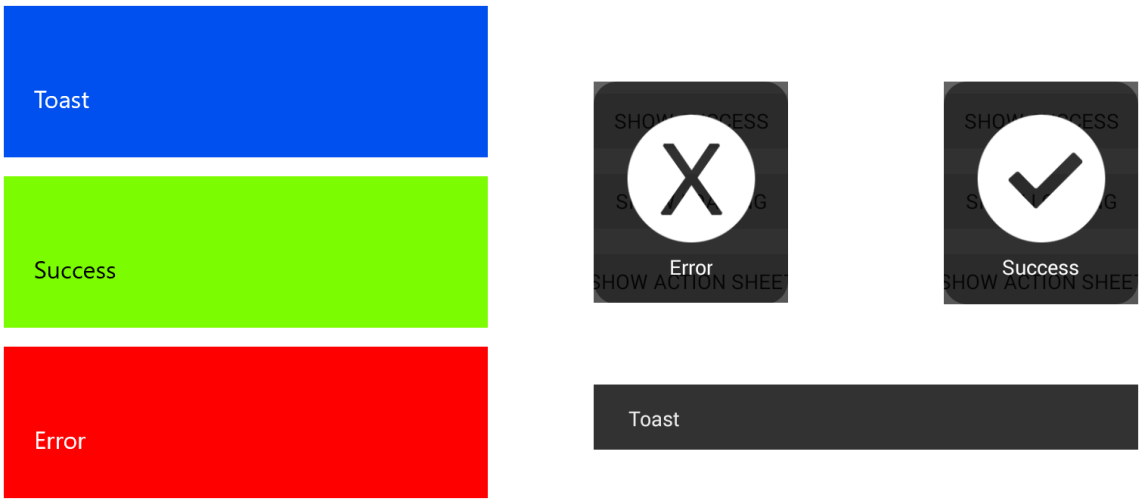
10.2 Acr.UserDialogs

Autorem Acr.UserDialogs je Allan Ritchie. Knihovna umožňuje použití mnoha druhů dialogů pro interakci s uživatelem, jejichž vzhled je platformě specifický. Pro správnou funkčnost je třeba nainstalovat příslušný balíček do všech platformě specifických projektů. V projektu pro Android je dále třeba do hlavní aktivity vložit kód `UserDialogs.Init(this)`. Pramenem podkapitoly je [2].

Některé typy dialogů lze konfigurovat pomocí speciálních objektů skládajících se z názvu dialogu a doplněným slovem „Config“. Dialogy získávající informaci pomocí interakci uživatele obsahují synchronní a asynchronní verzi, kdy jeden z nepovinných parametrů asynchronní verze je typu `CancelationToken`. Práce s knihovnou spočívá ve

volání statických metod třídy `UserDialogs` pomocí `UserDialogs.UserDialogs.Instance`. Dále budou uvedeny některé z množiny dostupných dialogů. V první části budou popsány dialogy výhradně synchronní, v druhé pak ty s asynchronní verzí.

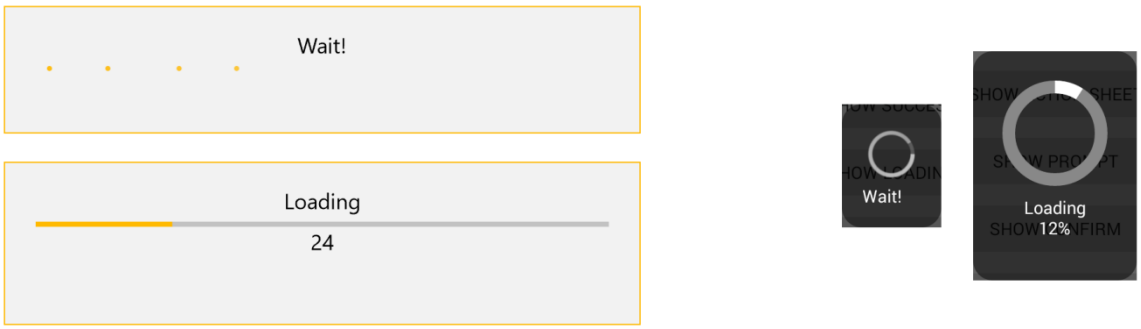
Zobrazení krátké informace je možné pomocí metod `Toast`, `ShowError`, `ShowSuccess` a `ShowImage` viz Obr. 10. Parametrem všech těchto metod je zobrazovaný text a volitelným parametrem je doba, po kterou má být tento text zobrazen (ve výchozím nastavení 2 sekundy). Na zařízeních s Windows 10 způsobí volání metod `Toast`, `ShowError` a `ShowSuccess` zobrazení obdélníku s textem v levém horním rohu obrazovky aplikace. Text lze specifikovat prvním parametrem každé z metod. Rozdíl mezi jednotlivými metodami je v barvě zobrazeného obdélníka. V případě použití metody Toast je barva stejná, jakou má uživatel zvolenou v systému. V případě metody `ShowError` je barva červená, v případě `ShowSuccess` je zelená. Metoda `ShowImage` není pro UWP podporována. Na zařízeních Android je metoda `Toast` podobná, pří použití metod `ShowSuccess` a `ShowError` se zobrazí ikona pod kterou je zobrazen text zadaný jako parametr metody. Ikona a prostředí Android symbolizuje úspěšnost či neúspěšnost operace místo barev použitých v UWP. V prostředí Android je metoda `ShowImage` funkční a zobrazuje zadaný text pod obrázkem specifikovaným parametrem.



Obr. 10 Výsledek volání metod ShowToast, ShowSuccess a ShowError na platformách Windows (vlevo) a Android (vpravo)

Metoda `ShowLoading` zobrazí přes celou obrazovku masku a znemožní uživateli interakci s aplikací. Zobrazí platformě specifickou animaci indikující probíhající činnost s textem pod ní. Prvním parametrem metody je zobrazovaný text a druhým je maska typu `MaskType`. Její nastavení určuje, barvu a průhlednost masky. Při použití v desktopové

aplikaci UWP však nemá nastavení masky vliv. Na stejném principu pracuje metoda `Progress`. Metoda progres může být volána s parametrem typu `ProgressConfig`. Metoda vrací instanci typu `IProgressDialog`, které lze nastaví procentuální kompletnost vykonávané akce pomocí vlastnosti `PercentComplete`, na jejíž změnu dynamicky reaguje zobrazená animace. Visuální vzhled dialogů vytvořených pomocí metod `ShowLoading` a `Progress` je patrný z Obr. 11.



Obr. 11 Výsledek volání metod ShowLoading a Progress na platformách Windows (vlevo) a Android (vpravo).

Animace vytvořené metodami uvedenými v minulém odstavci lze ukončit pouze voláním další metody. V případě `ShowLoading` se jedná o statickou metodu `HideLoading` třídy `UserDialogs`, v případě metody `Progress` pak o instanční metodu `Hide` instance typu `IProgressDialog`, vrácené metodou `Progress`. Na mobilních zařízeních Windows 10 je však možné ukončit blokující animaci pomocí zpětného tlačítka.

Metody s možností asynchronního volání slouží k získání jisté informace od uživatele. Po dokončení uživatelské interakce je hodnota navrácena. Všem kromě `ActionSheetAsync` je také možné předat konfigurační objekt.

Pro získání textu je k dispozici metoda `PromptAsync`. Po jejím vyvolání je zobrazen box s titulkem (vlastnost `Title`), zprávou (vlastnost `Message`), polem pro zadání textu, které může být předvyplněné (vlastnost `Text`) a jedním či dvěma tlačítky s texty (vlastnost `OkText` a případně `CancelText`). Vždy je zobrazeno potvrzující tlačítko, tlačítko rušící je zobrazeno pouze v případě nastavení vlastnost `IsCancellable` na hodnotu `true`. Nastavit lze i zástupný text (vlastnost `Placeholder`) a očekávaný typ textu (vlastnost `InputType`). Z vráceného objekt typu `PromtResult` lze zjistit, jaké tlačítko bylo stisknuté (vlastnost `Ok`) a zadaný text (vlastnost `Text`).

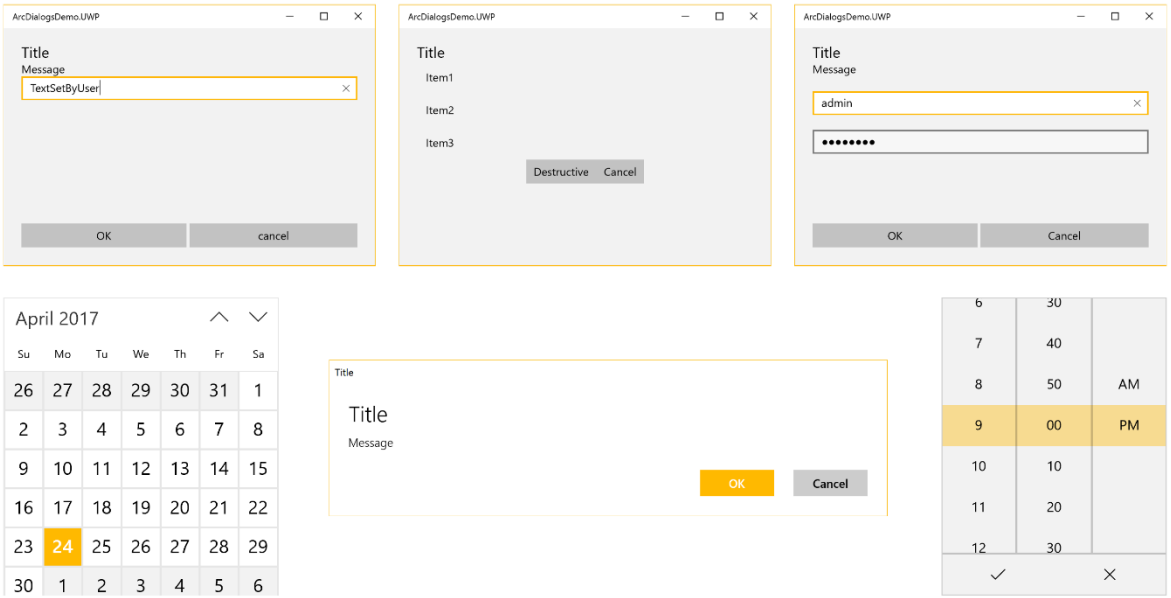
Pro zobrazení množiny tlačítek, kdy stisk libovolného z nich způsobí navrácení textu zobrazeného na něm, slouží metoda `ActionSheetAsync`. Parametry jsou titulek (`title`),

text destruktivního (`destructive`) a rušícího (`cancel`) tlačítka a pole řetězců reprezentující texty tlačítek, z nichž uživatel vybírá.

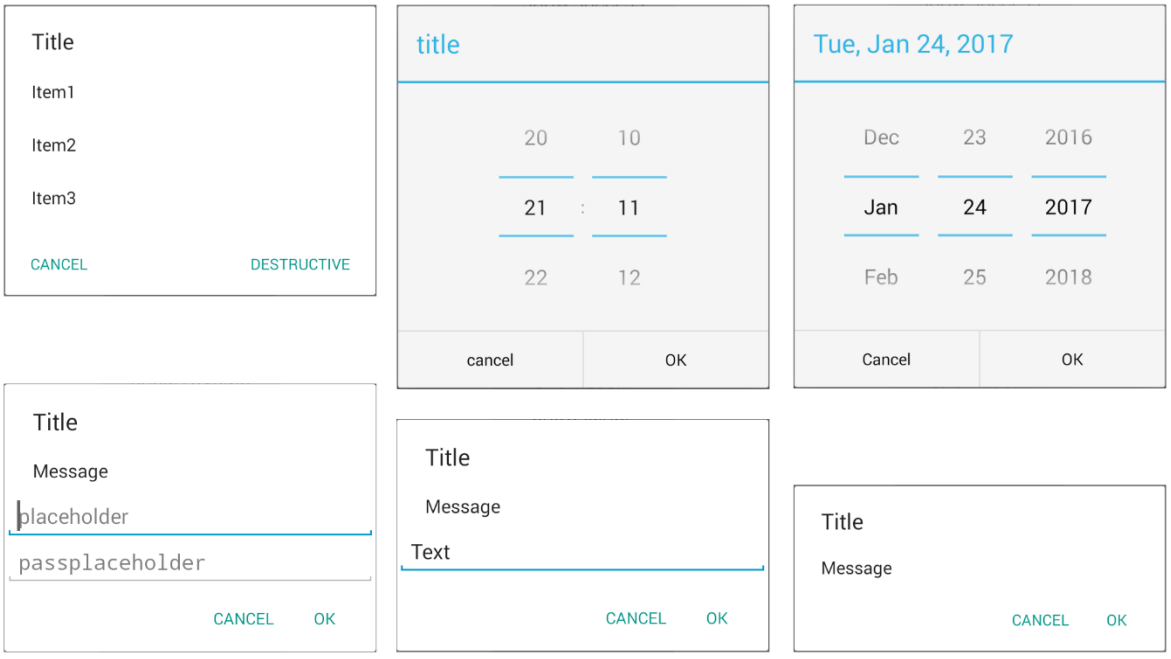
Metoda `ConfirmAsync` způsobí zobrazení boxu s titulkem (vlastnost `Title`), zprávou (vlastnost `Message`) a dvěma tlačítky, kdy jedno z nich je potvrzující s textem zadaným vlastností `OkText` a druhé z nich je rušící s textem zadaným vlastností `CancelText`. Jakmile uživatel stiskne jedno z tlačítek, je navrácena hodnota typu `bool` indikující, zda bylo stisknuto potvrzující tlačítko.

Acr.Userdialogs umožňuje také získávání data a času a to pomocí metod `DatePromptAsync` a `TimePromptAsync`. Oběma metodám lze konfigurovat titulek, positivní a negativní text tlačítek a přepínač indikující, zda má být rušící tlačítko zobrazeno. `TimePrompAsync` umožňuje navíc mimo jiné určit, zda má být zobrazen 24 hodinový formát (`Use24HourClock`) a minimální nastavitelný krok v minutách (`MinuteInterval`). `DatePrompAsync` zase umožňuje nastavení minimálního a maximálního nastavitelného data. Z návratových hodnot lze opět zjistit, zda bylo stisknuto potvrzující tlačítko (vlastnost `Ok`) a hodnotu data (vlastnost `SelectedDate`) či času (vlastnost `SelectedTime`).

Poslední asynchronní metodou je metoda `LoginAsync` sloužící pro zadání uživatelského jména a hesla. Po volání této metody je zobrazen box s titulkem, zprávou a dvěma elementy pro zadávání textu kdy horní z nich slouží k zadávání přihlašovacího jména (vlastnost `Login`) a spodní k zadávání hesla. `Login` lze přednastavit a pro přihlašovací jméno i heslo lze nastavit zástupný text (`LoginPlaceholder` a `PasswordPlaceholder`). Zadané znaky hesla jsou zastoupeny krycími znaky. Návratová hodnota obsahuje opět vlastnost `Ok` určující, zda bylo stisknuto potvrzovací tlačítko a vlastnosti `Login` a `Password` typu string obsahující zadané hodnoty. Na Obr. 12 lze vidět výše popsané dialogy zobrazené na platformě Windows a na Obr. 13 pak dialogy zobrazené na platformě Android.



Obr. 12 Výsledky volání metod PromptAsync, ActionSheetAsync, LoginAsync, DatePromptAsync, ConfirmAsync a TimePromptAsync v pořadí od levého horního rohu po pravý dolní roh volaných na zařízením s OS Windows.



Obr. 13 Výsledky volání metod ActionSheetAsync, TimePromptAsync, DatePromptAsync, LoginAsync, PromptAsync a ConfirmAsync v pořadí od levého horního rohu po pravý dolní roh volaných na zařízení s OS Android

10.3 sqlite-net-pcl

Multiplatformní podporu SQLite databází nabízí knihovna sqlite-net-pcl, jejímž autorem je Frank A. Krueger. Pro dosažení požadované funkcionality je třeba instalovat balíček do sdíleného projektu i do platformě specifických projektů. K dispozici je jak synchronní, tak asynchronní API. Vzhledem k faktu, že v aplikaci PT41 bylo využito pouze asynchronní API, nebude dále synchronní verze popsána. V této podkapitole bylo čerpáno z [20].

Pro ukládaná data jsou vytvořeny třídy, kdy jejich vlastnosti representují sloupce tabulky. Metody či privátní datové členy jsou při ukládání do databáze ignorovány. Pomocí atributů lze jednotlivým vlastnostem definovat, zda se jedná o primární klíč, zda má být použita automatická inkrementace, zda má být položka ignorována a další.

Před vlastním využitím databáze je třeba vytvořit objekt typu `SQLiteAsyncConnection`, jehož konstruktor obsahuje parametr typu `string` určující cestu k databázi. Instanční metody takto vytvořeného objektu umožňují asynchronní práci s databází, která je representována souborem, jehož umístění odpovídá cestě zadané výše uvedenému konstruktoru. Dotazování je možné pomocí řetězcové SQL (*Structured Query Language*) syntaxe za využití metod `ExecuteAsync`, `QueryAsync` a `ExecuteScalarAsync`. Funkce dalších metod vyplívá z jejich názvů a nebudou dále rozebírány. Zajímavou vlastností je `Table` generického typu `AsyncTableQuery`, s jejíž pomocí lze přistupovat k datům způsobem podobným LINQ (*Language Integrated Query*).

10.4 rda.SocketsForPCL

Síťovou komunikaci zajišťuje knihovna rda.SocketsForPCL, jejímž autorem je Ryan Davis. Pro umožnění komunikace je třeba přidat knihovnu do sdíleného projektu i do všech platformě specifických projektů, v nichž je navíc třeba nastavit příslušná oprávnění aplikace. Využít lze třídy `TcpSocketListener`, `TcpSocketClient`, `UdpSocketReceiver`, `UdpSocketClient` a `UdpSocketMulticastClient`. Aplikace PT41 využívá pro komunikaci protokolu TCP (*Transmission Control Protocol*) a je v ní využita pouze třída `TcpSocketClient`, proto následuje její podrobnější popis [19].

Třída obsahuje bezparametrický konstruktor a instanční metody `ConnectAsync`, `DisconnectAsync` a `GetConnectedInterfaceAsync`. Jelikož třída implementuje rozhraní `IDisposeble`, je k dispozici i metoda `Dispose`. Parametry metody `ConnectAsync` jsou IP adresa v řetězcovém formátu, port typu `int`, příznak určující, zda se má využít zabezpečené spojení (pomocí protokolu TLS (*Transport Layer Security*)) a objekt typu `CancellationToken`. Ostatní metody jsou bezparametrické. Návratovou hodnotou

metody `GetConnectedInterfaceAsync` je objekt typu `ICommsInterface` obsahují informaci o připojeném rozhraní [19].

Třída `TcpSocketListener` obsahuje také řadu vlastností. Pomocí `RemotePort` a `RemoteAddress` lze získat port a IP adresu protistrany. Vlastní komunikace probíhá zápisem či čtením objektů typu `Stream` zpřístupněných pomocí vlastností `WriteStream` a `ReadStream`. Z instančních metody třídy `Stream` budou využity asynchronní metody `WriteAsync`, jejíž parametry jsou `buffer` typu `byte[]` z kterého mají být data vyčtena, jejich počáteční index a délka. Poslední parametr je typu `CancellationToken`. Metoda `ReadAsync` má stejné parametry, oproti metodě `WriteAsync` jsou však data čtena z objektu typu `Stream` a uložena do parametru typu `byte[]`[19].

10.5 SkiaSharp

V prostředí Xamarin.Forms lze také využít bohatou grafickou 2D knihovnu Skia.Sharp. Balíček Skia.Sharp.Forms s jeho závislostmi je nutné instalovat do sdíleného projektu i projektů platformě specifických. Knihovna Skia je dostupná i pro jiné programovací jazyky a je využívána například společnostmi Google či Mozilla. S využitím této knihovny je možné vytvářet vlastní multiplatformní uživatelské elementy. Výše byl vysvětlen princip vytváření vlastních uživatelských elementů pomocí vlastních rendererů. Tento přístup má však proti knihovně Skia.Sharp několik nevýhod. V první řadě je třeba implementovat kód na každé z platforem, čímž se zvětšuje množství kódu a také požadavky na znalosti programátora. Navíc vytváření vlastních elementů na jednotlivých platformách může být v závislosti na typu elementu stejně složité, případně i složitější než v případě použití knihovny Skia.Sharp. Knihovna je značně rozsáhlá a budou rozebrány pouze základy práce s ní [8].

Grafické prvky lze kreslit na objekt typu `SKCanvas` neboli plátno. Tato třída obsahuje velké množství instančních metod umožňující čištění plátna, vykreslení tvarů, textů či obrázků na plátno a mnohé další. Barevnou informaci grafických objektů nese instance třídy `SkPaint`. Třída umožňuje prostřednictvím svých vlastností mimo jiné nastavení barvy, okrajů, šířky okrajů, antialiasingu, ale i vlastností týkajících se textu jako velikost písma nebo jeho styl. Je možné také definovat barevné filtry, transformace s využitím transformačních matic či průhlednost a mnohé další [8].

11 Aplikace PT41

Účelem aplikace PT41 je obousměrná komunikace s jednotkami PT41 za účelem nastavení či vyčtení konfigurace. Požadavkem na aplikaci je práce ve třech jazycích (čeština, angličtina a němčina). Mobilní i desktopová verze aplikace musí být schopna komunikovat s převodníkem PRE30 ETH/WIFI pomocí protokolu 802.11, desktopová verze bude navíc schopna komunikovat přes USB rozhraní pomocí PRE USB/RS232. Požadavkem je také uživatelsky přívětivé respozivní GUI funkční na platformách Android a Windows. Vstupy a výstupy jednotek jsou v aplikaci souhrnně označeny jako zóny.

Aplikace sestává z celkem patnácti aktivních a devíti neaktivních projektů. Projekty obsahující vytvořené uživatelské elementy jsou umístěny ve vlastních jmenných prostorech. Jeden z nich, (PT41.VisualControls) obsahuje elementy vytvořené pomocí vlastních rendererů či s využitím již existujících elementů. Druhý, nazvaný SkiaVisuals obsahuje projekty s elementy vytvořenými pomocí knihovny Skia.Sharp. Vlastní aplikační logika je umístěna v projektech, jejichž název začíná textem „PT41“ následovaným typem platformy. Těchto projektů je celkem šest, z nichž jeden je sdílen, a další jsou platformě specifické. Stejné množství a typy projektů obsahují také oba výše zmíněné jmenné prostory. Jmenný prostor PT41.VisualControls obsahuje navíc sdílený projekt pro všechny platformy Windows. Jsou tedy implementovány projekty pro všechny dostupné platformy. Projekty pro iOS, Windows 8.1 a WindowsPhone 8 jsou však neaktivní. V aplikaci jsou umístěny knihovny obsahující další funkcionalitu. Jedná se o knihovnu AppConstants zastřešující konstanty aplikace, ExtensionMethods obsahující rozšiřující metody, PT41.Models obsahující datové třídy reprezentující modely a Services obsahující třídy pro usnadnění práce s databází a další pomocné třídy.

11.1 Visuální elementy vytvořené pomocí Skia.Sharp

Jak již bylo zmíněno, elementy vytvořené pomocí knihovny Skia.Sharp se nachází v odděleném jmenném prostoru obsahujícím celkem šest projektů, z nichž nejobsáhlejším je projekt sdílený. Pro potřeby aplikace bylo pomocí Skia.Sharp vytvořeno celkem sedm visuálních elementů. Některé z nich je třeba přizpůsobovat změně velikosti okna, jiné mají rozměry konstantní. Nutné je také zajištění shodného vzhledu na zařízeních s různou hodnotu PPI. Všechny visuální elementy mají své nastavitelné vlastnosti zastřešené statickou vlastností pouze pro čtení typu `BindableProperty`, čímž se stávají možným cílem data bindingu. Každý vytvořený element má přepsanou metodu `OnMeasure`, v níž vrací ním vyžadovanou velikost. Tato velikost je uváděna v DIU.

Kromě tříd obsahujících implementaci visuálních elementů obsahuje sdílený projekt třídy `GraphicalHelper`, `GraphicalConstants`, `ViewGestures` a rozhraní `IDiuToPixelRatio`. Rozhraní `IDiuToPixelRatio` obsahuje definici bezparametrické metody `GetDiuToPixelRatio` s návratovou hodnotou typu `double`. Toto rozhraní je využito ve statické metodě třídy `GraphicalHelper` a každý platformě specifický projekt obsahuje třídu implementující toto rozhraní. Návratová hodnota udává počet pixelů obsažených v abstraktní jednotce DIU. Volání probíhá pomocí třídy `DependencyService` implementované v Xamarin.Forms. Třída `GraphicalHelper` dále obsahuje metody pro získání bodů ležících na části kružnice. Parametry jsou počet bodů, které mají být navráceny, souřadnice části kružnice, a její poloměr. Dalšími parametry jsou procentuální obvod kružnice a počáteční úhel ve stupních. Návratovou hodnotou je seznam objektů typu `SKPoint` ležících na specifikované kružnici.

Poslední metodou je `GetOvalPath` s návratovou hodnotou typu `SKPath` definující cestu tvořící ovál. Tato metoda využívá metody `GetCirclePartPoints` a proto má stejné parametry doplněné dalším typu `int` specifikujícím délku mezi půlkružnicemi na ovále. Třída `GrapicalConstants` obsahuje pouze konstantu `PXToTextSizeRation` typu `float` určující poměr velikosti textu uváděného knihovnou Skia.Sharp a skutečnou velikostí písma v pixelech.

Třída ViewGestures obsahuje visuální element vytvořený pomocí vlastních rendererů s přídavnou funkcionalitou pro obsluhu některých druhů interakce uživatele s možností získání dodatečných informací o pozici provedené interakce. Kód byl získán z [17] a mírně modifikován. S jeho pomocí lze registrovat události započetí stisku elementu, ukončení stisku elementu, potáhnutí po elementu, dlouhý stisk elementu a další.

11.1.1 Element vlastní switch

Prvním z implementovaných elementů, jejichž rozměry se nemění v závislosti na velikosti okna aplikace je vlastní switch implementovaný ve třídě `HugeSwitch`. Oproti elementu `Switch` implementované v Xamarin.Forms má vytvořený mnoho nových vlastností. Jedná se o nastavitelnou barvu pozadí, kterou lze nastavit separátně pro zapnutý (`OnColor`) a vypnutý (`OffColor`) stav udávaný vlastností `IsToggled` typu `bool`. Element navíc obsahuje pro oba stavy nastavitelné texty (`OnText` a `OffText`), jejich barvu lze také nastavit v závislosti na stavu elementu (`ActiveTextColor` a `InactiveTextColor`). Nastavit lze také barvu okraje (`StrokeColor`). Třída dále obsahuje konstanty udávající rozměry elementu a velikost písma v DIU a vlastnosti s přepočtenou hodnotu v pixelech. Element je tvořen objektem `SKCanvasView` uzavřeném v elementu `ViewGestures` pro zajištění možnosti detekce stisku.

V metodě `OnMeasure` je inicializován objekt typu `SKPaint` sloužící k vykreslení textu. Následně je s jeho pomocí (metoda `MeasureText`) vypočtena šířka obou textů i s nastaveným minimální horizontálním rozestupem mezi nimi (konstanta `MinTextsMarginHorizontalDiu`). Následně je vypočtena délka grafické části elementu. Požadovaná délka celého elementu je pak maximem z vypočtených hodnot. Je-li šířka textu větší než šířka grafického elementu, je nastaven offset, o který má být grafický element při vykreslení odsunut. Návratová hodnota obsahuje vypočtenou šířku a výšku.

Vlastní vykreslení elementu probíhá v metodě `PaintSurface`. V ní je nejprve pomocí výše zmíněné metody `GetOvalPath` třídy `GraphicalHelper` načtena cesta oválu reprezentující grafický element. Metodě je předána hodnota posunu v horizontálním směru vypočtená v metodě `OnMeasure`. Následně jsou vytvořeny objekty typu `SKPaint`, nesoucí barevné údaje okraje a vlastního oválu a obě části jsou vykresleny. Další objekty typu `SKPaint` jsou vytvořeny pro aktivní text, neaktivní text a kruh reprezentující rukojeť. Dále je na základě vlastností `IsToggled` vykreslen kruh buď na začátku elementu, nebo posunutý a jsou vykresleny texty se správně přiřazenou barvou textu. V metodě `ElementTapped` je určena pozice stisku a podle ní je rozhodnuto, zda byl stisk proveden v oblasti grafického elementu. Pokud tomu tak bylo, je změněna vlastnost `IsToggled` a volána metoda `InvalidateSurface` třídy `SKCanvas`, která způsobí překreslení elementu.

Obr. 14 Element vlastní switch

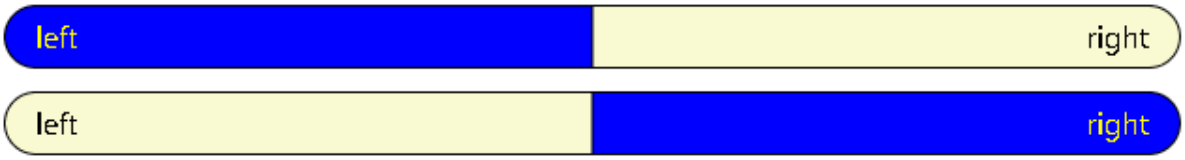
11.1.2 Přepínač a textový switch

Přepínač i textový switch jsou stejně jako vlastní switch určený k možnosti zadávání či zobrazování hodnoty typu `bool`.

Přepínač je oproti elementu vlastní switch podstatně menší a nemá dodatečné texty. Disponuje vlastnostmi `IsToggled` typu `bool`, jejíž význam je stejný jako v případě elementu switch a dále `OffColor` a `OnColor` udávající barvu elementu v zapnutém či vypnutém stavu. Visuální vzhled elementu je takový, že v neaktivním stavu je zobrazena kružnice zvolené barvy (`OffColor`) a ve stavu zapnutém má tato kružnice jinou barvu (`OnColor`) a je doplněna dvěma přímkami reprezentující znak „fajfka“ viz Obr. 15. Při stisku elementu je v metodě `Touched` analyzováno, zda byl proveden stisk v aktivní oblasti elementu a pokud ano, je negována hodnota vlastnost `IsToggled` a dojde k voláno metody `InvalidateSurface`.

Obr. 15 Element přepínač

Textový switch je svým visuálním vzhledem i vlastnostmi velice podobný elementu vlastní switch. Rozdílný je v tom, že podpůrné texty nejsou zobrazeny pod elementem, ale jsou jeho součástí. Stiskem elementu lze vybrat levou či pravou část elementu a nastavit tak vlastnost `IsLeftSelected`. Na stisk je také reagováno překreslením elementu. Nastavit lze levý a pravý text, barvu textu vybrané i nevybrané poloviny a jejich pozadí. Způsob vykreslení je podobný způsobu implementovaném v elementu vlastní switch s tím rozdílem, že element zabírá veškerou dostupnou šířku. Velikost písma textů je adaptována tak, aby součet šířek textů nepřesáhl dostupnou velikost.



Obr. 16 Element textový switch

11.1.3 Vlastní elementy typu slider

Xamarin.Forms obsahuje implementaci elementu slider. Jeho možnosti jsou však značně omezené. Třída `Slider` obsahuje pouze nastavitelné vlastnosti `Maximum`, `Minimum` a `Value` typu `double`. Jejich výchozí vzhled je navíc pro většinu aplikací nepřijatelný a jeho změna pomocí vlastních rendererů je velmi složitá. Dalším problémem elementu Slider je nemožnost reakce na ukončení interakce uživatele s tímto elementem. V průběhu interakce se může vlastnost `Value` měnit velice často a reagovat na každou její změnu může být značně neefektivní.

Z těchto důvodů byla knihovna SkiaVisuals doplněna o čtyři elementy typu slider. Všechny vytvořené elementy typu slider obsahují vlastnost typu `Command`, jehož metoda `Execute` je vyvolána po ukončení uživatelské interakce.

První dva elementy typu slider (`StepSlider` a `SliderWithSpecifiedValues`) mají oproti obyčejnému elementu slider implementován uživatelsky přívětivý vzhled a také nastavitelný krok. Krok lze v případě elementu `StepSlider` nastavit pomocí vlastnosti `Step` typu `double`. Element `SliderWithSpecifiedValues` umožňuje pomocí své vlastnosti `AvailableValues` typu `List<double>` určit nastavitelné hodnoty. Oba elementy mají možnost nastavení barvy manipulační rukojeti tvaru kruhu a pozadí za ní. Při stisku či tahu elementu je průběžně sledována stisknutá pozice a je ukládána do pomocného datového člene.

S její pomocí je vypočtena případná nová hodnota a pozice rukojeti. Následně dojde k překreslení elementu. Při vykreslení elementu je nejprve vykreslen obdélník barvy nastaveného pozadí. Na něj je pak nanesen obdélník aktivní barvy, který horizontálně začíná v bodě shodném se začátkem elementu a končí v bodě, v němž je provedena interakce uživatele. V posledním kroku je vykreslena rukojeť. Pokud uživatel ukončí svou manipulaci v místě, které neodpovídá žádné z nastavitelných hodnot, je po ukončení manipulace element znovu překreslen s nejbližší dostupnou hodnotou. Vzhled elementů `StepSlider` a `SliderWithSpecifiedValues` je shodný a je patrný z Obr. 17.

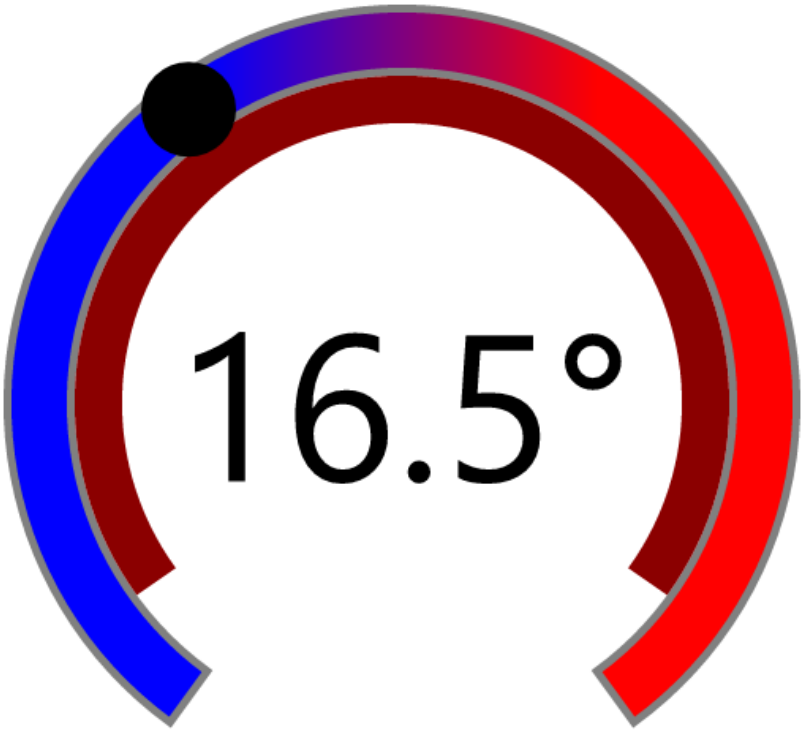
Obr. 17 Element StepSlider či SliderWithSpecifiedValues

Dalším z elementů typu slider je `RangeSlider` umožňující nastavit dvě hodnoty, které udávají rozsah. Visuální vzhled je podobný předchozím s tím rozdílem, že aktivní část je vykreslena mezi dvěma uživatelem vybranými body viz Obr. 18. Po stisku či potáhnutí elementu je doplněna logika určující, zda uživatel manipuluje s horní, či spodní hodnotou a je aktualizována příslušná souřadnice. Kromě všech vlastností uvedených u elementu `StepSlider` lze navíc nastavit maximum minimální hodnoty a minimum maximální hodnoty. Místo vlastnosti `Value` jsou u tohoto elementu k dispozici dvě vlastnosti určující spodní (`LowValue`) a horní (`HighValue`) nastavenou hodnotu.

Obr. 18 Element RangeSlider

Posledním elementem typu slider je kruhový slider. Interaktivní plocha u tohoto elementu obepíná část kružnice. Uprostřed je navíc zobrazen text dynamické velikosti s nastavitelnou barvou udávající vybranou hodnotu. Pod interaktivní oblastí je dodatečná oblast tvaru části kruhu, vylepšující grafický vzhled elementu. Obsaženy jsou všechny vlastnosti elementu `StepSlider`, navíc jsou však dispozici vlastnosti určující barvu textu (`FontColor`), barvu okraje interaktivní oblasti (`SelectableStrokeColor`) a barvu dodatečné oblasti (`DesignAreasColor`). Kruhový slider je navíc vybaven logikou, díky níž je schopen přizpůsobit svou velikost velikosti jemu dostupného prostoru. Funkcionalita obsluhy stisku a vykreslení je podobná s elementem `StepSlider`, zde je však nutná dodatečná logika pro převod uživatelských pohybů, které nemusí přesně korespondovat interaktivní oblasti do pohybu rukojeti, který probíhá vždy v této oblasti. Změna velikosti dostupného místa je detekována pomocí vlastnosti `Width`, která je dostupná jako člen vlastnosti `Info` objektu `SKPaintSurfaceEventArgs`, který je předán metodě `PaintSurface` jako druhý parametr. Takto získaná hodnota je potom srovnána s hodnotou

minulou, která je zálohována pomocí datového člene třídy. Je-li hodnota rozdílná, pak došlo ke změně velikosti obrazovky a je nutné vypočítat nový poloměr elementu a aktualizovat polohu rukojeti. Kruhový slider je zobrazen na Obr. 19.



Obr. 19 Element kruhový slider

11.2 Ostatní vytvořené visuální elementy

Kromě elementů vytvořených pomocí grafické knihovny Skia.Sharp byly vytvořeny i další uživatelské elementy a to jak pomocí již existujících elementů tak pomocí vlastních rendererů. Tyto elementy jsou umístěny ve jmenném prostoru `PT41.VisualControls`. Výše bylo uvedeno, že vytváření elementů pomocí již existujících elementů snižuje množství duplicitního kódu.

V aplikaci se často vyskytuje trojice elementů `TimePicker`, `StepSlider` a `Label`, kdy element `Label` zobrazuje hodnotu vybranou pomocí elementu `StepSlider`. Z tohoto důvodu byl vytvořen element `TimePickerWithSlider` sjednocující tyto elementy.

Další element vytvořený z již existujících elementů je `ImageSwitch`. Element se skládá z elementu `Image` s přídavnou funkcionalitou pro obsluhu stisku. Po stisku je navíc volána metoda `Execute` objektu typu `Command` definovaného vlastností `StateChangedCommand`. `ImageSwitch` slouží stejně jako element `Switch` k zadávání informace typu `bool`. Tato informace je udržována vlastností `IsOn` typu `bool`. Po stisku elementu dojde k negaci této vlastnosti. V závislosti na `IsOn` je zobrazován obrázek nastaven jako pozitivní (`OnImage`) či

negativní (`OffImage`). Tento element by mohl nahradit elementy vlastní switch a jednoduchý přepínač rozebrané výše. Existuje však několik důvodů, proč využívat elementy vytvořené pomocí knihovny Skia. Prvním z nich je rozšiřitelnost. Rukojeť elementu vlastní switch ve své současné konfiguraci je dle jeho stavu vykreslena na pravé, či na levé straně. Pokud by však v budoucnu bylo nutné vytvořit animaci pro přechod mezi stavy, nejednalo by se o obtížnou úlohu. V případě elementu `ImageSwitch` je tato možnost vyloučena. Další nevýhodou je množství různě velkých obrázků, potřebných pro jeho zobrazení na všech platformách. Například platforma Android obsahuje sedm složek, v nichž jsou uloženy obrázky různých rozměrů. Takový vývoj je náročný z hlediska času stráveného vytvářením obrázků. Následně se obrázky stávají součástí programu a zvyšují tak jeho paměťové nároky. Xamarin.Forms umožňuje vložní obrázků do sdíleného projektu, čímž se výše uvedené požadavky snižují, nastává však další podstatný problém a tím je neurčitost. Každá platforma totiž zobrazí obrázek se stejným počtem pixelů jinak velký. Je sice možné omezit rozměry elementu `Image` vlastnostmi `HeightRequested` a `WidthRequested`, výsledek ovšem nemusí být kvalitní. V takovémto případě využije každá z platforem vlastní algoritmus na škálování obrázku, čímž dojde ke ztrátě jeho barevné informace.

Další vytvořený element, `EntryStyled` je element `Entry` obohacený speciálním stylem. Takovýto způsob je poměrně složitý nicméně možnosti nastavení vlastnosti `Style` základních elementů z Xamarin.Forms jsou značně omezené. Pokud je třeba měnit visuální vzhled elementů v jejich určitých stavech (neaktivní, stisknutý, označený), je situace ještě horší a je třeba vytvořit styl pro každou platformu separátně. Naopak relativně jednoduchá je tato úloha v případě práce s programem Blend pro Visual Studio. Výsledné styly jsou pak aplikovatelné pouze na elementy implementované pro platformu Windows. Tohoto principu bylo využito při tvorbě elementu `EntryStyled`, kdy zařízením s OS Windows byly upraveny jejich speciální stavy.

11.3 Knihovna Services

Knihovna Services obsahuje čtyři třídy a jedno rozhraní. Dvě třídy společně s rozhraním jsou umístěny se speciálním jmenném prostoru `PT41.Services.Database` a slouží pro zjednodušení práce s databází. Další dvě třídy jsou umístěny ve jmenném prostoru `PT41.Services.Helpers` a jedná se o statické pomocné třídy. V první části bude rozebrán jmenný prostor `PT41.Services.Database`.

Rozhraní nazvané `IDbFileGetter` obsahuje jednu metodu s názvem `GetLocalFilePath`. Účelem této metody je získání platformě specifické cesty, kde bude vytvořen soubor s databází. Metoda má pouze jeden parametr typu `string`. Jedná se o název souboru. Návratová hodnota, která je také typu `string` by měla obsahovat cestu do lokálního

úložiště zařízení a končit zadaným jménem souboru. Platformě specifické projekty toto rozhraní implementují ve svých třídách tak, aby bylo následně možné využití třídy `DependencyService` z Xamarin.Forms.

Cesta vytvořená tímto způsobem je parametrem konstruktoru první ze tříd knihovny Services. Jedná se o generickou třídu `DbService` s omezením generického parametru na typ `ModelBase` či jeho potomka s bezparametrickým konstruktorem. Instance této třídy usnadňuje práci s databází, kdy generický typ representuje tabulku databáze. `DbService` navíc podporuje ukládání načtených instancí do paměti pomocí objektu generického typu seznam (cache), kdy generický parametr je shodný s generickým parametrem třídy. Při požadavku na prvek databáze je vrácen prvek ze seznamu cache, čímž je značně zkrácena přístupová doba. Při požadavku na zápis či aktualizaci prvku je nejprve aktualizován seznam cache a až poté je provedeno uložení do souboru s databází. Všechny databázové operace probíhají asynchronně. V konstruktoru třídy je vytvořeno spojení s databází, které je uloženo do privátního datového člene a je naplněn seznam cache.

První instanční metodou třídy je `GetCount` s návratovou hodnotou typu `int` udávající počet prvků v tabulce. Další metodou je `SaveItemsAsync` s parametrem typu `List<T>`, kdy jednotlivé položky jsou stejného generického typu jako třída. Zadané prvky jsou nejprve dle jejich identifikátoru roztříděny na ty, které mají být do databáze vloženy, a na ty, které mají být aktualizovány. Dojde také k aktualizaci či vložní prvku do seznamu cache. Vlastní vkládání či aktualizace prvků probíhá pomocí instančních metod objektu typu `SQLiteAsyncConnection` vytvořeného v konstruktoru. Metoda `SaveItemAsync` pracuje stejným způsobem, ukládá či aktualizuje však pouze jeden prvek. Získání všech prvků tabulky umožňuje metoda `GetAll`. V důvodu urychlení operace vrací tato metoda načtený seznam cache. Pro mazání prvku z tabulky slouží metody `DeleteItemsAsync` a `DeleteItemAsync`. První z metod slouží pro smazání seznamu prvků, které jsou jejím parametrem a druhá pro mazání jediného prvku. V obou případech je opět příslušným způsobem aktualizován seznam cache.

Poslední třídou jmenného prostoru `PT41.Services.Database` je `DbServiceOfModelsWithParentId`, která rozšiřuje funkcionalitu třídy `DbService` o metodu `GetItemsFromParent`. Jejím generickým parametrem může být pouze třída `ModelWithParent` či její potomek s bezparametrickým konstruktorem. Návratovou hodnotou metody `GetItemsFromParent` jsou všechny prvky, jejichž cizí klíč odpovídá tomu zadanému v parametru metody.

Jmenný prostor `PT41.Services.Helpers` obsahuje třídy `ArrayHelper` a `PropertiesHelper`. Třída `PropertiesHelper` obsahuje pouze jednu generickou metodu `SetToRange`, jejímž úkolem je nastavení hodnoty proměnné předané referencí tak,

aby byla v rozmezí zadaném druhým a třetím parametrem. Generický typ je omezen na typy implementující rozhraní `IComparable`. Návratová hodnota je typu `bool` a udává, zda byla hodnota předané proměnné mimo zadaný rozsah. Vlastní logika spočívá ve využití metody `CompareTo` rozhraní `ICompareble`. Je-li předávaná hodnota menší než hodnota minimální či větší než hodnota maximální, dojde k jejímu nastavení na minimum respektive maximum a je vrácena logická hodnota `true`. V opačném případě je vrácena hodnota `false` a s předávanou hodnotou není vykonána žádná operace.

Třída `ArrayHelper` obsahuje množinu generických statických metod, které lze využít pro práci s daty representovanými typem pole. Metoda `GetSubArray` je metoda, jejímž prvním parametrem je pole generického typu. Účelem metody je vrátit část pole zadanou indexem a délkou. V případě, že zadané parametry přesahují rozměry pole, je vrácena hodnota `null`. Generická metoda `AreArraysEqual` generického typu vrací hodnotu typu `bool`. Generický typ je omezen na třídy implementující rozhraní `IEquatable`. Parametry jsou dvě pole zadaného generického typu. Pokud mají pole stejnou délku a všechny prvky v nich jsou shodné (návratová hodnota metody `Equals` je `true`), je vrácena hodnota `true`. V opačném případě dojde k navrácení hodnoty `false`. Metoda `ByteArrayToHexString` převádí pole s prvky typu `byte` na řetězec a slouží pro výpis trasovacích hlášek. V metodě je využita třída `BitConverter`. Další generická metoda `Concat` slouží ke konkatenaci polí. Parametrem je dvourozměrné pole generického typu s označením `params`. Jedná se tedy o metodu, jíž lze předávat proměnný počet parametrů typu pole generických prvků. Jednotlivá pole zadaná parametrem jsou do výsledného pole umístěna za sebe v pořadí, v jakém byla metodě předána a výsledné pole je navráceno. Metoda `HasDataOnTheEnd` využívá výše uvedených metod `GetSubArray` a `AreArraysEqual`. Její návratová hodnota určuje, zda pole zadané prvním parametrem obsahuje na svém konci prvky zadané parametrem druhým. Metoda `AnyStartsWithData` má tři parametry. Prvním z nich je pole prvků generického typu, druhým je seznam polí tohoto typu a třetím je seznam prvků s hodnotami typu `int`. Účelem metody je zjistit, zda některá z polí zadaných druhým parametrem má na svém počátku prvky stejné jako pole zadané prvním parametrem. Poslední parametr udává seznam pozic v poli, na kterých není třeba shodnost prvků kontrolovat. Pro porovnávání polí využívá tato metoda přetíženou metodu `AreArraysEqual` třídy `ArrayHelper`. Tato metoda postupně maskuje všechny prvky na zadaných indexech a volá metodu `AreArraysEqual` přetíženou s dvěma parametry.

11.4 Knihovna Models

Knihovna Models obsahuje datové třídy reprezentující tabulky databáze. Navíc obsahuje dvě abstraktní třídy, které jsou bázovými třídami datových tříd. První bázovou třídou je

ModelBase. Její jedinou vlastností je Id typu int udávající primární klíč. Proto je tato vlastnost označena atributem [SQLite.PrimaryKey]. Třída implementuje rozhraní INotifyPropertyChanged, aby bylo možné použít instance datových tříd pro data binding. Druhou abstraktní třídou je třída ModelWithParent, která rozšiřuje třídu ModelBase o vlastnost ParentId typu int. Třída slouží k uložení prvků, s vazbou na prvek v jiné tabulce. Touto vazbou je cizí klíč specifikovaný vlastností ParentId typu int.

V knihovně jsou obsaženy tři datové třídy. Všechny obsahují nastavení specifická vstupu jednotky PT41. Všechny také dědí z jedné z výše popsaných abstraktních tříd. V popisu jednotek bylo uvedeno, že jednotlivým vstupům je možné nastavit program sestávající z šesti dvojic čas-teplota pro každý den v týdnu. Datovou representací jsou třídy DayDataModel a TimeTeperatureEntryModel.

TimeTemperatureEntryModel dědí z třídy ModelWithParent a obsahuje dvojici vlastností Time typu TimeSpan a Temperature typu double. Tyto vlastnosti udávají dvojici čas-teplota. Cizím klíčem je primární klíč třídy DayDataModel. Vlastnost Id lze nastavit pomocí metody SetId s parametry udávajícími identifikátor rodiče (cizí klíč) a index nastavovaného prvku v rámci rodiče.

DayDataModel representuje jeden den v programu. Obsahuje vlastnost výčtového typu DayOfWeek určující den. Třída také obsahuje seznam prvků typu TimeTemperatureEntryModel, které jsou však uloženy do oddělené tabulky a proto je seznam označen atributem [Ignore]. V seznamu je implementováno řazení jeho prvků dle času. Při inicializaci seznamu je události ProperyChanged každého z prvků přiřazena metoda KeepItemsSorted. Tato metoda má za úkol třídit prvky seznamu dle jejich času. Navíc musí být zajištěno, že prvky se stejným časem budou mít nastavenu stejnou teplotu. Po úspěšném seřazení je vyvolána událost ItemsSorted, jejímž prvním parametrem je seznam obsahující změněné položky. Aby nedocházelo vlivem změny prvků seznamu k cyklické aktualizaci všech položek, je použit datový člen _isUpdating typu bool udávající, zda probíhá aktualizace prvků či nikoliv. Pokud ano, je metoda ihned opuštěna. V opačném případě dojde k nastavení tohoto příznaku a vyprázdnění seznamu změněných položek. Instance, která byla změněna přímo, je obsažena v parametru sender metody KeepItemsSorted. Tato instance je ihned přidána do seznamu změněných prvků a je zjištěn její index v rámci seznamu všech prvků. Následně jsou procházeny prvky takové, jejichž index je menší než index změněné položky a současně je čas v nich nastavený větší či roven času změněné položky. Těmto položkám je nastavena hodnota času i teploty stejná, jako je hodnota změněného prvku a jsou přidány do seznamu změněných položek. Následně jsou procházeny ty prvky, jejichž index je větší než změněný prvek a jejich čas je menší či roven času tohoto prvku. Těmto položkám je také nastaven čas a teplota změněného prvku a jsou

přidány do seznamu změněných prvků. Na závěr metody je vyvolána metoda `OnItemsSorted`, které je předán seznam změněných prvků a příznak udávající aktualizaci prvků je nastaven na hodnotu `false`.

Poslední datovou třídou je `ZoneModel` obsahující zbylá data specifická pro konkrétní vstup či výstup. Tato třída již nemá žádné závislosti v podobě cizích klíčů a je přímým dědicem třídy `ModelBase`. Její vlastnost výčtového typu `ConnectedDeviceType` určuje typ vstupu či výstupu. Možnými hodnotami jsou `Thermostat`, `Pump`, `Boiler`, `Sensor` a `Nothing`. Na základě této vlastnosti je nastavena vlastnost `InputTypeImgSource` typu `ImageSource` udávající ikonu zvoleného typu zařízení. `InputTypeImgSource` není do tabulky ukládán. Další vlastností, která není do tabulky uložena je seznam s prvky typu `DayDataModel` obsahující program dané zóny. Prioritu zóny ke spínání výstupu určuje vlastnost `HasPriority`. Zvolený mód (automatický či manuální) je určen vlastností `IsManuMode`. Pomocí `IsOn` se rozlišuje aktuální stav zóny. Každá zóna má také své nastavitelné jméno uložené ve vlastnosti `Name`. V neposlední řadě jsou přítomny vlastnosti obsahující údaje o teplotě. `CurrentTemperature` obsahuje teplotu změřenou jednotkou. Dále jsou uloženy požadované teploty v režimu auto (`AutoTemp`) a manu (`ManuTemp`).

Stav výstupů je uložen ve vlastností `IsOutputReleSwitchedOn`. Výše bylo uvedeno, že od verze 12.04 je možné nastavit každý vstup tak, aby byla její teplota korigována pouze útlumovou teplotou nastavenou globálně. V případě využívání této funkcionality je nastavena vlastnost `IsTempManagedByKeepingTemp` na hodnotu `true`. Třída `ZoneModel` obsahuje dodatečné metody, pomocí nichž je možné integrovat data přijatá z jednotkou PT41. Některé z nich pracují s bitovými flagy. Pomocí logických operátorů je z předaného bytu maskováním vyextrahována informace typu `bool`, která je pak uložena. Podrobnější popis vyžaduje detailní popis komunikačního protokolu a bude uveden dále.

11.5 Knihovny ValueConverters AppConstants a ExtensionMethods

Zbylé tři knihovny, jejichž názvy jsou uvedeny v názvu této podkapitoly, nejsou příliš rozsáhlé, a proto jsou sloučeny do jedné podkapitoly. K jejich vytvoření došlo zejména z důvodu možnosti budoucího rozšíření aplikace.

Knihovna `ValueConverters` obsahuje pouze dvě třídy. První z nich je generická třída nazvaná `BoolToObjectConverter`. Tato třída implementuje rozhraní `IValueConverter` a tudíž obsahuje metody `Convert` a `ConvertBack`. K dispozici jsou také dvě vlastnosti zadaného generického typu (`TrueObject` a `FalseObject`). V metodě `Convert` je na základě parametru `value` určeno, který z objektů má být navrácen. Tato třída je příjemným zobecněním převodu vlastnosti typu `bool` na vlastnost libovolného typu a může

tak nahradit mnoho specifických tříd implementujících rozhraní `IValueConverter` (například třídu `BoolToVisibilityConverter` často využívanou v prostředí UWP). Druhá třída, `DeviceTypeToImageSwitchConverter` slouží k určení viditelnosti elementu `ImageSwitch`, který určuje možnost nastavení, zda bude daná zóna řízena útlumovou teplotou. Tato hodnota je nastavitelná pouze v případě, je-li připojeno čidlo, případně je-li vstup nezapojen.

Pouze jednu třídu obsahuje knihovna `ExtensionMethods`. Tato třída s názvem `StringExtensions` rozšiřuje funkcionalitu proměnných typu `string`. Jediným třídním členem je metoda `GetSubstringWithLength`. Tato metoda je podobná metodě `Substring` třídy `string`, čili vrací část zadaného textu, kdy délka navráceného textu je udávána druhým parametrem. Při zadání délky větší než je délka řetězce je však v případě metody `Substring` vyvolána výjimka. Třetím parametrem metody `GetSubstringWithLength` je zástupný znak, kterým je text v takovém případě doplněn a k vyvolání výjimky tak nedojde.

Knihovna `AppConstans` obsahuje aplikační konstanty. V knihovně se nacházejí dvě třídy. Třída `DataConstants` obsahuje konstanty, které určují rozsah proměnných využitých v aplikacích, kroky elementů typu slider použitých při jejich zobrazení a také je zde implementován výčtový typ `ConnectedDeviceType`. Třída `VisualConstants` obsahuje konstanty týkající se vzhledu visuálních elementů. Jsou zde definovány odsazení elementů či stránek, často využívané barvy a také cesty k souborům obsahujícím obrázky.

11.6 Sdílený projekt PT41

Většina funkcionality se nachází ve sdíleném projektu typu PCL nazvaném PT41. Celkem je zde obsaženo šest složek a tři další třídy. Jejich popis následuje v podkapitolách.

11.6.1 Jmenný prostor PT41.Resources

Ve složce Resources jsou obsaženy tři soubory. Každý z nich obsahuje texty přeložené do jednoho z aplikací podporovaných jazyků. Podrobnější popis způsobu implementace vícejazyčné aplikace je uveden v 11.7.

11.6.2 Jmenný prostor PT41.DefaultData

Jak již bylo uvedeno, aplikační data jsou za pomoci datových tříd dědících z třídy `ModelBase` ukládány do SQLite databáze. V případě, že uživatel spustí aplikaci poprvé, neexistuje žádná možnost jak zjistit současnou konfiguraci jeho jednotek PT41. Aby měla aplikace uživatelsky přívětivý vzhled i po prvním spuštění, jsou v takovémto případě reálná

data nahrazena vygenerovanými. Ke generování těchto dat slouží třída `DefaultDataGenerator` z jmenného prostoru PT41.DefaultData.

Třída obsahuje tři metody, kdy každá z nich slouží pro generování prvků typu některé z datových tříd, které byly popsány výše. Nezbytnou vlastností těchto metod je správné nastavení vlastnosti `Id` a případně `ParentId` vytvářených elementů. Nejnižší hodnotou identifikátoru je vždy v rámci dané tabulky hodnota jedna. Privátní klíče nejsou inkrementovány automaticky. Tato funkcionalita není implementována z toho důvodu, aby nebylo nutné uvádět další číselné identifikátory udávající pozici daného datového prvku v kolekcích. Dříve bylo uvedeno, že jednotkám PT41 lze globálně nastavit minimální a maximální teplotu. Tato hodnota musí být brána v potaz i v případě generování výchozích dat. Třída `DefaultDataGenerator` nastavuje všem prvkům minimální nastavitelnou teplotu.

Metoda `GetZones` slouží k vytvoření výchozích dat typu `ZoneModel`. Jejími parametry jsou počet prvků, které mají být vygenerovány a identifikátor první z generovaných zón. V těle metody jsou ve smyčce tvořeny instance typu `ZoneModel` a jsou přidávány do kolekce. Minimální požadovaná teplota v režimu auto i manu je nastavena na hodnotu nastavitelného minima, jméno zóny je nastaveno v závislosti na jazykových preferencích zařízení. Je-li identifikátor zóny roven číslu 7, je jako typ připojeného zařízení zvoleno čerpadlo v případě, rovnosti identifikátoru s číslem 8 je zvolen jako typ zařízení kotel. Ostatní zóny mají zvolen typ teplotní čidlo. Ve výchozím nastavení se tedy předpokládá, že první připojenou jednotkou je jednotka PT41-M. Návratovou hodnotou je naplněný seznam.

Metoda `GetDayDataModels` slouží pro generování seznamu prvků typu `DayDataModel`. Počet navrácených prvků je vždy sedm, pro každý den v týdnu jeden. Parametrem metody je zóna, jíž budou vygenerované objekty náležet. V těle metody jsou v cyklu přidávány prvky typu `DayDataModel` do kolekce. Identifikátor zóny předané parametrem je každému prvku uložen do vlastnosti `ParentId` a vlastnost `Id` je dopočítána v instanční metodě `SetId` třídy `DayDataModel`, které se kromě čísla zóny předává také počítadlo. Návratovou hodnotou je opět naplněný seznam.

Pro generování seznamu objektů typu `TimeTemperatureEntry` slouží poslední metoda `GetTimeTempEntryes`. Jejím parametrem je objekt typu `DayDataModel` kterému budou prvky přiřazeny. Vygenerovaných položek je vždy šest. Princip generování prvků je stejný jako u předchozí metody.

11.6.3 Jmenný prostor PT41.DataSenders

Aplikace musí být schopna komunikovat pomocí 802.11 a v případě desktopové verze zařízení s OS Windows také pomocí USB. Tuto funkcionalitu zprostředkovávají třídy jmenného prostoru `PT41.DataSenders`, který obsahuje dvě třídy a dvě rozhraní.

Rozhraní `IDataSender` obsahuje metody nutné k navázání a ukončení spojení a také k odeslání a čtení dat. Všechny tyto metody jsou asynchronní. `IDataSender` implementuje třída `TcpIpDataSender`, sloužící k zajištění komunikace s využitím technologie 802.11 a protokolu TCP. Třída je implementována pomocí návrhového vzoru jedináček. V konstruktoru je vytvořen objekt typu `TcpSocketClient`, který byl popsán výše. Metody z rozhraní `IDataSender` tento objekt využívají. Je využito verze bez použití protokolu TLS a každé z metod je předáván objekt typu `CancelationToken` pro zajištění maximální doby trvání dané metody. V případě vzniku výjimky je tato výjimka vyhozena ke zpracování vyšší vrstvě. Všechny platformě specifické projekty mají upravena práva tak, aby bylo možné komunikaci provozovat (Internet (Client) a Private Networks (Client & Server) pro UWP a INTERNET v případě zařízení s OS android).

`IDataSender` implementuje i rozhraní `IUsbDataSender`. `IUsbDataSender` implementuje třída `UsbDataSender` platformě specifického projektu PT41.UWP. Proto má tento projekt dále upravena práva. Ke komunikaci dochází pomocí objektu typu `SerialDevice`. Tento objekt je získán pomocí statické metody `FromIdAsync` této třídy. Dále je tento objekt adaptován nastavením modulační rychlosti, parity a počtu datových bitů a stop bitů. Komunikace probíhá s využitím vlastností `InputStrem` a `OutputStream` tohoto objektu. `InputStrem` a `OutputStream` jsou při práci zabaleny do objektů typu `DataReader` respektive `DataWriter`, které se používají ke čtení či zápisu. [18]

Každé z metod je opět předáván parametr typu `CancelationToken` pro nastavení časování. Pro využití komunikace pomocí USB je v projektu PT41 třeba získat rozhraní `IUsbDataSender` s využitím `DependencyService` z Xamarin.Forms.

Poslední třídou rozebíraného jmenného prostoru je třída `DataSender`, která obaluje funkcionalitu tříd implementujících `IDataSender`. Parametrem konstruktoru třídy je instance typu `IDataSender`, pomocí níž bude komunikace probíhat. Nejdůležitější metodou je `SendMessageGetResponseAsync`, která slouží k odeslání zprávy a navrácení odpovědi. V případě nastavení druhého parametru metody na `true`, je navíc odpověď uložena do databáze nebo do úložiště definovaného vlastností `Properties` třídy `App` (o způsobu ukládání dat je pojednáno v 11.6.6). Pro některé zprávy z komunikačního protokolu platí, že délka jejich správné odpovědi není stejně dlouhá jako délka jejich chybové odpovědi. Proto je třeba použít metodu pro čtení dat třídy typu `IDataSender` několikrát pro jednu zprávu. Vzhledem k tomu, že metody pro čtení a zápis jsou asynchronní, může být požadavků na tyto

operace více současně. Knihovna rda.Socket sice neumožní čtení instance typu `Stream` více vláknům současně, problémem je však čtení zprávy sestávající z více částí. Jelikož po přečtení první části první zprávy prvním vláknem je ihned započato čtení druhým vláknem, které je ve frontě a čte tak místo druhé zprávy druhou část zprávy první. Aby se tomuto zabránilo, je ve třídě `DataSender` implementován statický objekt typu `Semaphore`. Před započetím zápisu zprávy je nejprve čekáno na uvolnění semaforu. Poté dojde k odeslání zprávy a následnému přijetí celé odpovědi a až poté je semafor uvolněn. K nekonzistentnímu čtení tak nemůže dojít.

Výše bylo uvedeno, že třídy implementující `IDataSender` žádným způsobem neošetřují výjimky a jejich ošetření je ponecháno vyšší vrstvě, kterou je právě třída `DataSender`. Při zachycení výjimky je odesílání zprávy zopakováno. Maximální počet opakování udává konstanta `MaxRetriesCount` třídy `ProtocolConstants`. K uvolnění semaforu dojde až po vyčerpání všech pokusů, nebo správném přijetí odpovědi. Pokud je počet opakování převršen, je odeslána testovací zpráva, která již není opakována. Není-li obdržena odpověď ani na tuto zprávu, je spojení ukončeno a je vyhozena výjimka. V opačném případě dojde pouze k vyhození výjimky, jelikož spojení je aktivní a není jej třeba rušit.

11.6.4 Komunikační protokol

Komunikační protokol udává formáty zpráv. Každá zpráva je definována svou hlavičkou, tělem a zápatím. Hlavička určuje typ zprávy, tělo obsahuje vlastní data a zápatí je shodné pro všechny zprávy a slouží k detekci jejího konce. Pokud je třeba získat data z jednotky, je hlavička jiná než v případě zasíláni dat do jednotky. Jednotka PT41 nikdy komunikaci neinicializuje, pouze zasílá odpovědi na obdržené zprávy. Tyto odpovědi mají stejný formát jako vysílané zprávy. V případě, že je zpráva neznámá či dojde k jiné chybě, posílá jednotka PT41 speciální chybovou zprávu. V opačném případě jsou vyslána vlastní data, pokud o ně bylo žádáno, nebo je odeslána potvrzující zpráva.

První skupinou zpráv jsou zprávy obsahující data, jejichž datový obsah je délky jeden byte, nazvané jako konstantní zprávy. Pomocí těchto zpráv je možné číst či zapisovat například útlumovou teplotu, zvolený režim, minimální a maximální nastavitelnou teplotu, korekci teploty, konstanty spojené s režimy regulace typu hystereze i typu PI. Čísla s desetinnou řádovou čárkou je třeba před odesláním příslušným způsobem upravit, aby vznikla čísla celá. Typickou úpravou je násobení dvěma (teplota je nastavitelná s krokem 0.5). Některé zprávy vyžadují pro získání dat zaslání více individuálních zpráv. Pokud například uživatel zadá správné heslo v případě využití komunikace pomocí 802.11 nebo je připojen pomocí USB, může heslo měnit. Heslo se odesílá jako konstantní zpráva, kdy jeden byte representuje jedno písmeno hesla a další byte representuje čítač určující, kolikátý znak v pořadí je odesílán.

Program jednotlivých zón je odesílán zprávou se speciální hlavičkou. Jeho přenos je obousměrný. Ve zprávě je mimo jiné uvedeno číslo vstupu, ke kterému náleží. Vlastní tělo má délku 84 bytů a je v něm obsaženo šest dvojic čas-teplota pro každý den v týdnu. Komunikační protokol neumožňuje zasílání dat pouze pro konkrétní den či konkrétní dvojici čas-teplota. Proto je při každé změně programu nutné odeslat celou zprávu.

Další zprávy slouží k přenášení dat pouze v jednom směru. Každá z nich je opatřena vlastní hlavičkou. Mezi zprávy, které jsou zasílány pouze do zařízení, patří zpráva nastavující čas, zpráva pro informaci o ukončení komunikace a zpráva zasílající data specifická pro vstupy a výstupy. Mezi tato data patří teplota v režimu auto a manu a také další, odesílané formou pomocných flagů. Pomocné flagy určují nastavený mód (auto či manu), zapnutí či vypnutí výstupu, prioritu daného vstupu, řízení vstupu útlumovou teplotou a typ připojeného zařízení (termostat nebo čidlo). Zpráva zasílající specifická data pro vstupy obsahuje vždy data všech vstupů a není možné posílat data separátně pro vybraný vstup. Formát je takový, že počítadlo určuje typ přednášených dat (teplota manu, teplota auto či pomocné flagy) a následuje 32 bytů hodnot, vždy jeden byte pro jeden vstup či výstup. Maximální počet vstupů a výstupů je 32. Pokud je počet připojených vstupů a výstupů menší, je přesto třeba posílat zprávu s tělem délky 32. Některá data pak budou jednotkou ignorována.

Podobná zpráva je dostupná i v opačném směru (z PT41 do zařízení). Tato zpráva je však dále rozšířena o dvě nové položky nastavitelné čítačem. Jedná se o změřenou teplotu a další pomocné flagy určující typ připojené jednotky a stav výstupů.

Další zprávou odesílající data ve směru ze zařízení PT41 je zpráva obsahující čítač. Tento čítač jednotka inkrementuje tehdy, když dojde k externímu nastavení určitých hodnot. Čítač je uložen také v aplikaci a slouží k určení, zda došlo ke změně konfigurace od minulého připojení zařízení (například z jiného zařízení). Zpráva, jejíž odeslání způsobí inkrementaci tohoto čítače, implementuje rozhraní `ICounterChanger`. Jednotka také odesílá zprávy informující o počtu připojených jednotek a verzi firmware.

Implementaci tohoto protokolu tvoří třídy jmenného prostoru `PT41.CommunicationProtocol`. Aplikace vždy zasílá zprávu representovanou instancí třídy dědící z abstraktní metody `Pt41MessageBase`. Mezi vlastnosti třídy `Pt41MessageBase` patří `Header`, `Value` a `Footer` obsahující hlavičku tělo a zápatí zprávy. Další vlastnosti určují délku odpovědi a to jak bezchybné (`CorrectResponseLength`), tak chybové (`IncorrectResponseLength`). Směr přenosu dat je určen pomocí vlastnosti `IsAskingForData` a vykonaný počet pokusů odeslání zprávy je obsažen ve vlastnosti `RetriesCount`. Abstraktní metoda `GetResponse` slouží k propojení zprávy s vhodnou odpovědí typu `Pt41Response`.

Úlohou tříd dědících z `Pt41MessageBase` je pak naplnění hlavičky a těla, implementace metody `GetResonse` a případně naplnění chybových odpovědí. Zprávy mohou také implementovat rozhraní `IMessageAbleToSend` data. Rozhraní definuje jednu metodu `LoadData`. V této metodě je načteno tělo zprávy z uložených aplikačních dat.

Výše bylo zmíněno, že některé logické zprávy se skládají z více jednotlivých zpráv. Tyto zprávy dědí z třídy `MultiMessage`. Její vlastnosti jsou `IsError` určující zda došlo k chybě, `IsAskingForData` určující směr zasílání dat a `ExpectedResponsesCount` udávající počet fyzických zpráv, ze kterých se daná logická zpráva skládá. Třída také implementuje dvě abstraktní metody. První z nich, `GetMessages` slouží k získání seznamu fyzických zpráv. Druhá z nich, `GetResponse` vytváří objekt typu `MultiMessageResponse`, obsahující logickou odpověď na základě seznamu zadaných fyzických odpovědí.

Pro každou zprávu je ve jmenném prostoru PT41.CommunicationProtocol.Responses vytvořena třída reprezentující odpověď. Stejně jako v případě zpráv existuje abstraktní třída `Pt41Response`, kdy všechny odpovědi jsou jejími potomky. Její vlastnosti jsou podobné třídě `Pt41MessageBase`. Místo vlastnosti určují směr zasílání dat, však obsahuje vlastnost `IsContainingData` určující, zda daná odpověď obsahuje data nebo se jedná pouze o potvrzení. Úkolem tříd dědících z `Pt41Response` je naplnit vlastnost `IsContainingData` a implementovat metodu pro případné uložení dat.

Výše bylo uvedeno, že některé zprávy získávají data z aplikačních dat. Odpovědi mají naopak schopnost data uložena v jejich těle do aplikačních dat ukládat. Takové odpovědi implementují rozhraní `ISavingDataSync`. Jiné odpovědi obsahují data, která mají být uložena do databáze, přičemž k ukládání dochází asynchronně. Tyto odpovědi implementují rozhraní `ISavingDataAsync`. `ISavingDataSync` i `ISavingDataAsync` obsahují pouze jedinou metodu nazvanou `SaveData`, v případě `ISavingDataSync` je její návratová hodnota `void`, v případě `ISavingDataAsync` je návratová hodnota typu `Task`.

Odpovědí na zprávu representovanou potomky třídy `MultiMessage` je instance třídy dědící z `MultiMessageResponse`. Vlastnost `ExpectedInnerResponsesTypes` obsahuje seznam typů, které by měla daná odpověď obsahovat. Další abstraktní metoda `LoadDataFromInnerResponses` slouží k načtení dat z jednotlivých odpovědí. Odpověď je vyhodnocena jako chybná, je-li libovolná z fyzických odpovědí chybná, není-li těchto odpovědí požadované množství nebo jejich typy neodpovídají zadaným typům.

Součástí jmenného prostoru `PT41.CommunicationProtokol` jsou také další třídy. První z nich, `CommunicationData` je datová třída obsahující vlastnosti nutné k sestavení spojení a odesílání dat. Jedná je se o IP adresu, port, heslo a objekt typu `DataSender`. Instance této třídy je využita v druhé ze tříd `AllConfigHelper`, sloužící k odeslání či přijetí veškeré konfigurace z nebo do jednotky PT41. Třída obsahuje dvě asynchronní metody

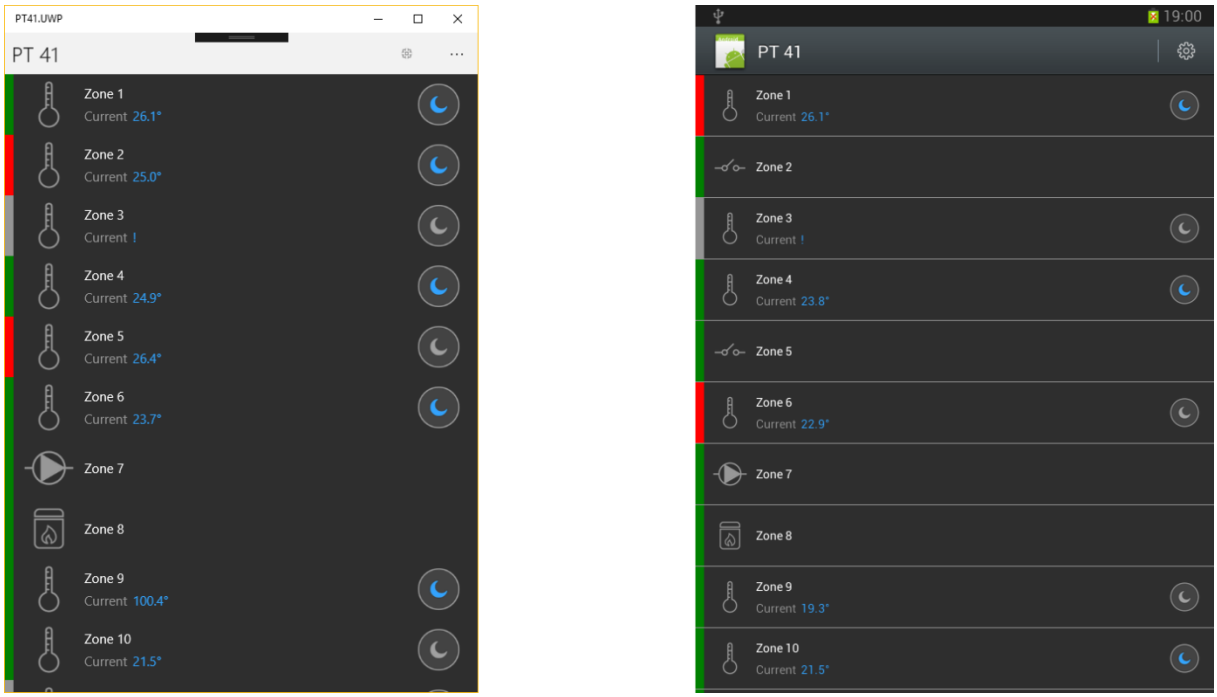
`GetDataFromDevice` a `SendAllToDevice`. V jejich tělech dochází k načtení seznamu potřebných zpráv. V případě odesílání zpráv do jednotky PT41 jsou těla zpráv načtena z aplikačních dat pomocí metody `Load` rozhraní `IMessageAbleToSendData`. V případě získávání konfigurace dojde následně k uložení přijatých dat pomocí metod `SaveData` rozhraní `ISaveDataSync` respektive `ISaveDataAsync`.

Třída `CommunicationHelper` obsahuje metody spojené s konverzí dat z podoby zobrazované do podoby využívané komunikačním protokolem. Konstanty spojené s komunikací jsou zase obsaženy ve třídě `ProtocolConstants`. Jsou zde uloženy pozice jednotlivých prvků ve zprávách, délka zpráv a jejich odpovědí a dále například maximální doba pro sestavení spojení, odeslání a přijímání zprávy, či maximální počet pokusů o odeslání zprávy. Mimo to jsou zde uloženy výchozí hodnoty některých proměnných.

11.6.5 Stránky a jejich view-modely

Aplikaci tvoří čtyři stránky. View-modely jsou jim přiřazeny pomocí jmenných konvencí a využití návrhového vzoru dependency service knihovny prism. Všechny view-modely stránek implementují rozhraní `INavigationAware` pro možnost obsluhy životního cyklu navigace. Všechny view-modely dědí z třídy `ViewModelBase`. Jelikož je při navigaci na stránku vytvořena vždy nová instance view modelu, je nutné po navigaci načíst stav z aplikačních dat. K tomu slouží abstraktní metoda `RestoreState`, volaná v metodě `OnNavigatedTo`.

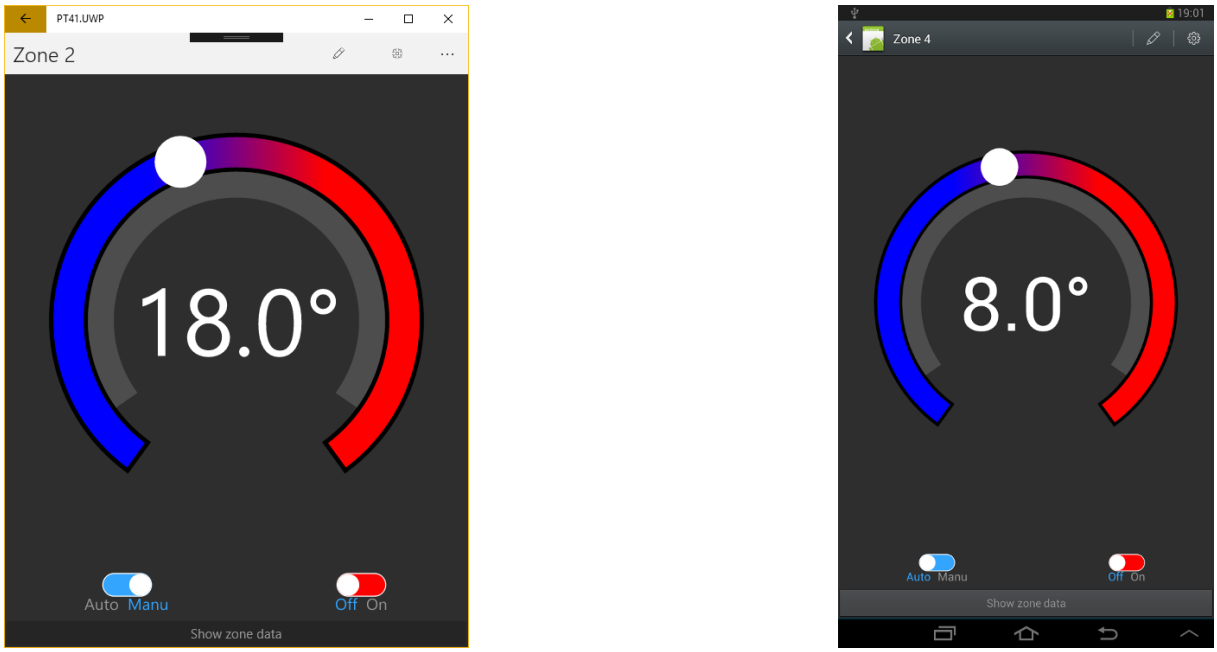
První stránka `MainPage` slouží k zobrazení seznamu připojených vstupů a výstupů, jejich stavů a případně detekovaných teplot. Levý bok každého prvku seznamu obsahuje barevný proužek definující stav výstupu, kdy zelená barva definuje sepnutý stav, červená rozepnutý stav a šedá nedefinovaný stav. Ve středu je zobrazen konfigurovatelný název vstupu a pod ním detekovaná teplota, která je zobrazena v případě připojeného teplotního čidla. Nezapojený vstup je indikován teplotou nastavenou na hodnotu “!“. Stiskem obrázku vpravo lze aktivovat či deaktivovat řízení vstupu pomocí útlumové teploty. Hlavička stránky umožňuje navigaci na stránku nastavení. Po stisku prvku reprezentujícího vstup, k němuž je připojeno teplotní čidlo nebo je nepřipojen je provedena navigace na stránku detailu zóny, `ZoneDetailPage`. Visuální vzhled stránky MainPage lze vidět na Obr. 20.



Obr. 20 Visuální vzhled stránky MainPage

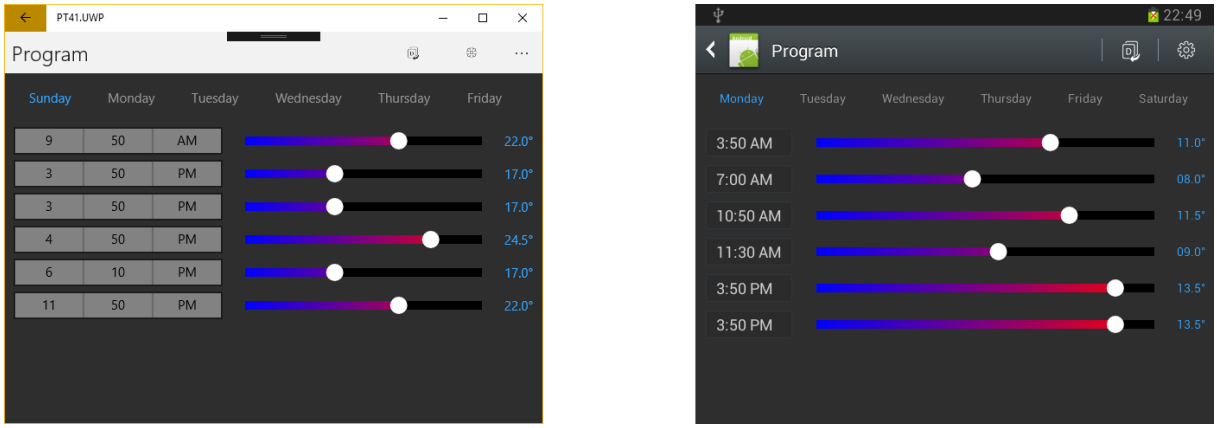
View-model hlavní stránky nejprve získá počet objektů typu `ZoneModel` dostupných v databázi. Pokud je tento počet roven nule, jsou vygenerovány výchozí data pomocí třídy `DefaultDataGenerator`. V opačném případě dojde k načtení dat z databáze. Načtená data jsou pak vložena do kolekce, která je na hlavní stránce zobrazena. Při navigaci na detailní stránku je jako parametr předán vybraný prvek typu `ZoneModel`.

`ZoneDetailPage` obsahuje informace o vybrané zóně. Na této stránce lze měnit mód zóny (auto/manu), nastavovat požadovanou teplotu ve vybraném módu a také nastavovat stav výstupu. Hlavička stránky obsahuje stejné položky jako hlavní stránka, navíc ale obsahuje tlačítko, po jehož stisku se zobrazí dialog umožňující změnit název vybrané zóny. Ve spodní části stránky je tlačítko umožňující navigaci na stránku umožňující konfiguraci programu zóny (`ZonesTablePage`). View model stránky načítá zónu z navigačních parametrů. Následně na základě jím nastavení režimu (auto či manu) nastaví požadovanou teplotu. Při navigaci na stránku `ZonesTablePage` je do navigačního slovníku vložen parametr obsahující vybranou zónu. Vzhled stránky je patrný na Obr. 21.



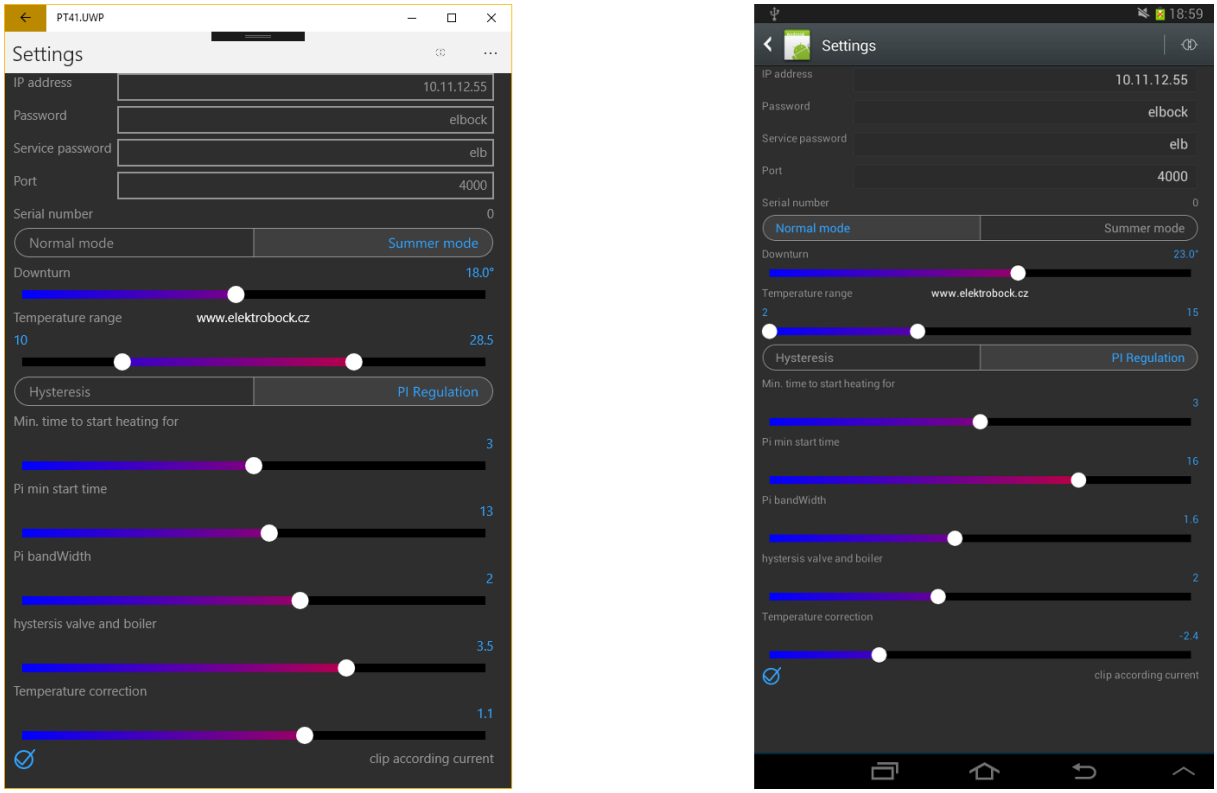
Obr. 21Visuální vzhled stránky ZoneDetailPage

Stránka `ZonesTablePage` obsahuje v horní části seznam všech dnů v týdnu, viz Obr. 22. Po výběru dne je zobrazena tabulka obsahující dvojice čas, požadovaná teplota. Tato konfigurace definuje program tak, jak byl uveden výše. Hlavička obsahuje stejné hodnoty jako hlavička hlavní stránky, navíc je zde další tlačítko, po jehož stisku je možné vybrat jeden den v týdnu, do něhož bude zkopírována konfigurace současně vybraného dne. Ve view modelu `ZonesTablePageViewModel` je po navigaci vyextrahována předávaná zóna a její program representovaný seznamem prvků typu `DayDataModel`. Třída `DayDataModel` automaticky řadí své prvky typu `TimeTemperatureEntryModel` a úspěšné seřazení je doprovázeno vyvoláním události `ItemsSorted`. Na tuto událost `ZonesTablePageViewModel` reaguje uložením změněných prvků typu `TimeTemperatureEntyModel` do databáze a odesláním příslušné zprávy jednotce PT41.



Obr. 22 Visuální vzhled stránky ZonesTablePage

Poslední stránkou je stránka nastavení (`SettingsPage`). V horní části této stránky je možné nastavit položky související se síťovým nastavením PT41. Jedná se o IP adresu, heslo pro 802.11 komunikaci, servisní heslo a port. Heslo pro 802.11 komunikaci je třeba nastavit při přístupu k zařízení pomocí protokolu 802.11. V případě použití rozhraní USB není heslo potřeba, jelikož je použita metoda autentizace předmětem, kdy předmětem je jednotka PT41. Hlavička stránky obsahuje tlačítko, které slouží k připojení či odpojení od zařízení. Vzhled stránky SettingsPage se všemi dostupnými nastaveními lze pozorovat na Obr. 23.



Obr. 23 Visuální vzhled stránky SettingsPage

Ve view modelu `SettingsPageViewModel` jsou v metodě `RestoreState` načteny všechny proměnné z aplikačních dat. Jejich počet je závislý na tom, je-li aplikace v servisním módu. Při požadavku na připojení k jednotce je nejprve zjištěno, zda je aplikace spuštěna na desktopové verzi zařízení s operačním systémem Windows. Pokud ano, je zobrazen dialog, v němž si uživatel vybírá způsob připojení (802.11 nebo USB). Následuje vytvoření objektu implementujícího `IDataSender`. V případě USB komunikace je použito přístupu `DependencyService` z Xamarin.Forms. Takto vytvořený objekt je zabalen do instance typu `DataSender`, který je uložen do komunikačních dat (typu `CommunicationData`), s jehož pomocí probíhá následná komunikace. V případě správně zadaného servisního hesla je aplikace přepnuta do servisního režimu a na stránce se zobrazí všechny další globální nastavení popsaná výše. Po nastavení libovolné vlastnosti uživatelem dochází k aktualizaci

aplikačních dat či databáze. Pokud je zařízení PT41 připojeno, je navíc odeslána příslušná zpráva.

Pomocné metody pro vykonání těchto funkcionalit obsahuje třída `App` v souboru App.xaml.cs projektu PT41. Tato třída obsahuje metody popsané při rozboru knihovny prism. V metodě `RegisterTypes` jsou zaregistrovány stránky k navigaci doplněné stránkou `NavPage` typu `NavigationPage`, na niž je navigováno ihned po spuštění aplikace a jejím obsahem je nastavena hlavní stránka. Aplikační třída dále obsahuje komunikační a aplikační data (typu `CommunicationData` a `ApplicationData`). Mezi její vlastnosti patří také instance tříd dědících z `DbService` sloužící pro ukládání dat do databáze. Je zde implementována i funkcionalita pro odeslání zprávy, kdy při neúspěšném odeslání dojde k informování uživatele. Metoda `ActualiseMinMaxTemp` slouží k zajištění korektnosti dat v databázi při změně minimální či maximální konfigurovatelné teploty. Výše bylo několikrát uvedeno, že data jsou čtena či zapsána do aplikačních dat. Aplikační data jsou obsažena ve vlastnosti `Properties` typu `IDictionary` třídy `App`. Klíčem tohoto slovníku je řetězec a hodnota je typu `object`. Vlastnosti uložené do této kolekce jsou platformě specificky uloženy a jsou konzistentní i po ukončení a opětovném spuštění aplikace. Nadstavbu tohoto slovníku tvoří třída `AppProperties` rozebraná níže.

11.6.6 Další třídy obsažené v projektu

Kromě výše rozebraných se v projektu PT41 vyskytují tři další třídy. Maximální a minimální nastavitelná teplota a informace, je-li aktivován servisní mód, jsou uloženy v datové třídě `ApplicationData`. Pokud došlo ke snížení maximální nastavitelné teploty nebo zvýšení minimální nastavitelné teploty, je vyvolána metoda `ActualiseMinMaxTemp` třídy `App` sloužící k zajištění korektního stavu databáze.

Další třída, `DbHelper` slouží k zjednodušení práce s databází. Obsahuje pouze jednu metodu `GetZonesWithDataFromDb`. Parametr je typu byte a udává počet zón, které se mají načíst. Metoda získá z databáze příslušné množství objektů typu `ZoneModel`. Navíc jsou načteny jejich seznamy prvků typu `DayDataModel`, kterým jsou dále načteny seznamy prvků typu `TimeTemperatureEntryModel`. Naplněný seznam prvků typu `ZoneModel` je pak návratovou hodnotou metody.

Poslední třída, `AppProperties` usnadňuje práci s aplikačními daty ve slovníku `Properties` třídy `App`. S pomocí této třídy lze k datům uloženým v tomto slovníku přistupovat jako k běžným vlastnostem jazyka C#. Navíc je zajištěna funkcionalita naplnění slovníku výchozími hodnotami a zajištěno správné typování prvků. Pro každý prvek, který má být uložen ve slovníku je vytvořena dvojice prvků, kdy první z nich je typu `string` a udává klíč v aplikačním slovníku a druhý je typu ukládaného elementu a jeho hodnota je výchozí

hodnotou. Pro každý prvek je také vytvořena statická vlastnost. Její metoda `set` nastavuje zadanou hodnotu do aplikačního slovníku, s klíčem definovaným ve výše popsané dvojici. Metoda `get` nejprve určí, zda je daný prvek ve slovníku a pokud ano, je jeho hodnota navrácena. V opačném případě je do slovníku vložen příslušný prvek s výchozí hodnotou a ta je také navrácena.

11.7 Podpora vícejazyčnosti aplikace

Pro podporu vícejazyčnosti byl využit nástroj ResXManager z [16]. Tento nástroj automaticky vytváří soubory s příponou resx, pro každý jazyk jeden. Tyto soubory nástroj umísťuje do složky Resources sdíleného projektu. Za běhu aplikace dochází k výběru použitého souboru na základě uživatelem vybrané jazykové sady. Nastavit lze výchozí jazyk, který je použit v případě, že pro uživatelem nastavenou sadu není žádný soubor vytvořen. Texty jsou zadávány v tabulce, kdy jejím prvním sloupcem je klíč a každý další sloupec obsahuje text specifický pro daný jazyk uvedený v hlavičce tabulky. Další funkcí nástroje je možnost uložení či načítání dat do souboru formátu xlsx. K dispozici je i automatický překlad z libovolného jazyka. Přeložené položky lze před uložením editovat.

Přístup k položkám uloženým v souboru s příponou resx proveden pomocí statických vlastností pouze pro čtení třídy, která vzniká ve stejném adresáři jako soubor s příponou resx. Vlastnosti jsou nazvány stejně, jak byly nazvány klíče v tabulce nástroje ResXManager.

11.8 Ladění a testování aplikace

Ladění multiplatformních aplikací zahrnuje testování programu na všech dostupných platformách. Na každé z platforem je vhodné testovat na co možná nejširším spektru zařízení. V této podkapitole budou uvedeny rozdíly v možnostech ladění multiplatformních aplikací vůči universálním aplikacím v prostředí Visual Studio a budou rozebrány možnosti připojení fyzických i virtuálních zařízení.

11.8.1 Ladění uživatelského rozhraní

Důležitým nástrojem pro vývoj aplikací obsahujících GUI je grafický designer. Při vytváření UWP aplikace umožňuje integrovaný designer Visual Studia v reálném čase sledovat změny kódu jazyka XAML a zobrazit výsledky do okna, které reprezentuje některé z fyzických zařízení, mezi kterými lze jednoduše přepínat. Tímto způsobem je možné testovat vzhled aplikace na různých zařízeních bez jejího spuštění. Pro vývoj UWP aplikací je navíc k dispozici speciální nástroj nazvaný Blend for Visual Studio. Tento program umožňuje konfiguraci stavů elementu (stisknutý, označený a další) a jejich vzhledu. Všechny tyto stavy

zobrazuje separátně v designeru, který je součástí tohoto nástroje. Blend lze využít také k vytváření animací a má mnohé další funkce. Při vývoji Xamarin.Forms nebyla do listopadu roku 2016 oficiálně dostupná ani jedna z výše uvedených variant (kromě beta verze). Bylo možné použít nástroj Gorilla Player dostupný z [12]. Tento nástroj je nutné nainstalovat na cílené zařízení a pak je na něm možné zobrazit tvořený kód jazyka XAML. Zařízení, na němž je kód zobrazen může být připojeno v lokální síti nebo pomocí USB. V listopadu 2016 byl uvolněn oficiální nástroj Xamarin.Forms Previewer s integrací do Visual Studia. Tento nástroj umožňuje zobrazování kódu jazyka XAML v reálném čase. Podporované platformy jsou však pouze Android a iOS. Zobrazit lze zařízení typu telefon či typu tablet v obou možných orientacích (na výšku či na šířku). Nelze však zvolit konkrétní zařízení, jedná se pouze o volbu typu. K zobrazení nedojde, pokud neexistuje bezparametrický konstruktor třídy aplikace, což je výrazný problém při tvoření aplikací vzorem definovaným knihovnou prism.

11.8.2 Využívání funkcí Visual Studia

Visual Studio nabízí pomocí svých oken možnosti detekce stavu proměnných (Locals), sledování proměnných (Watch), zobrazování výstupu (Output), výpis řetězce volaných metod (Call Stack) a další. Je také možné využít okno Immediate Window a s jeho pomocí za běhu programu vykonávat dodatečný kód do tohoto okna zadaný. Visual Studio inteligentně zobrazí případné výjimky včetně jejich popisu a dalších údajů. Všechny tyto funkce jsou při použití Xamarin.Forms podporovány při testování na zařízeních s OS Windows i s OS Android.

V ladícím režimu je ve Visual Studiu vykonávaná operace označena šipkou. Potáhnutím této šipky je možné měnit sled vykonávaných instrukcí a to dopředně i zpětně. Tato funkcionalita je však v Xamarin.Forms dostupná pouze v případě ladění UWP aplikace.

11.8.3 Dostupná virtuální zařízení

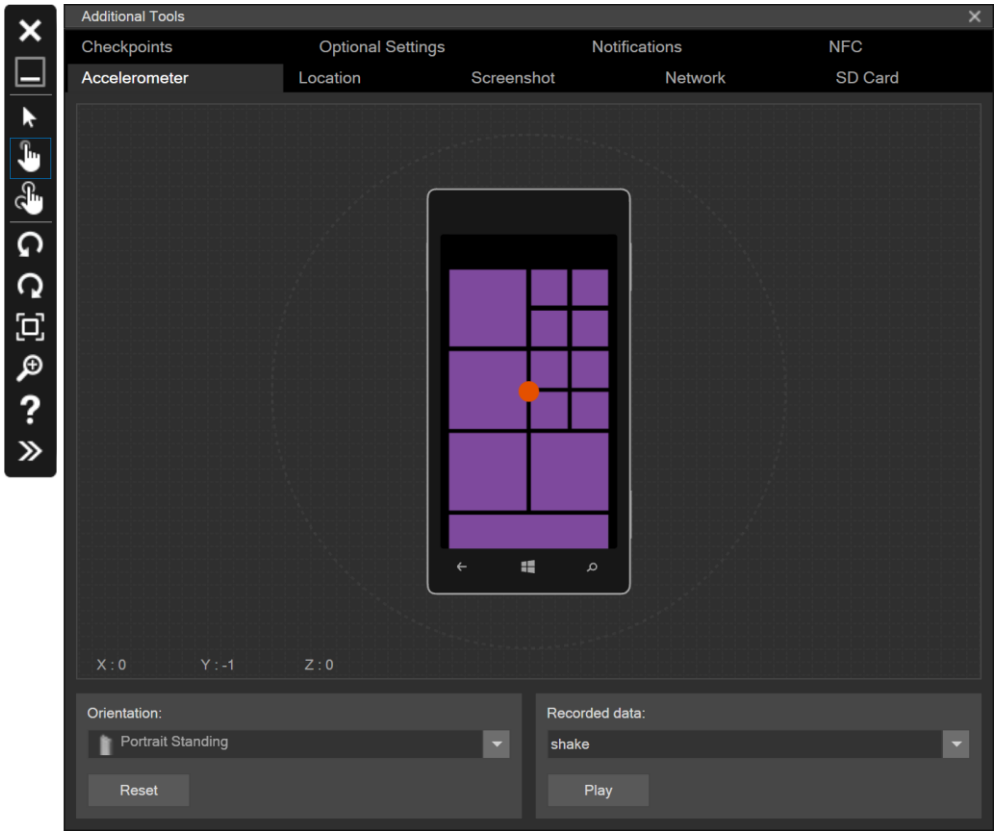
Virtuální zařízení jsou dostupná pro OS Android i pro UWP. Při zvolení UWP projektu jako výchozího je možné spustit aplikaci na vývojovém zařízení, pomocí mobilního emulátoru, pomocí připojeného fyzického zařízení nebo pomocí simulátoru.

Mobilní emulátory zařízení pro UWP lze stáhnout z [23]. Tyto emulátory lze řídit pomocí panelu nástrojů zobrazeného vedle nich. S pomocí těchto nástrojů lze simulovat dotyky uživatele, měnit orientaci zařízení a zvětšovat či zmenšovat zařízení. Jsou dostupná i pokročilá nastavení. Zde je možné mapovat soubor na disku jako SD (*Secure Digital*) kartu zařízení, řídit připojení k síti, konfigurovat NFC (*Near Field Communication*), pořizovat snímky obrazovky a provádět další operace dle vlastností emulovaného zařízení viz Obr. 24.

Při využití možnosti připojení fyzického zařízení je nutné, aby toto zařízení bylo připojeno ke stejné síti (volba Remote Machine), případně aby bylo připojeno USB kabelem (volba Device).

Při volbě Local Device, je aplikace nainstalována na vývojové zařízení a následně je na něm spuštěna. Volba simulátor zajistí spuštění aplikace na virtuálním zařízení typu tablet s konfigurací podobnou vývojovému zařízení. U emulovaného zařízení je dostupný panel nástrojů podobný tomu uvedenému u mobilních emulátorů.

K ladění programů pomocí zařízení s OS Android lze využít připojeného zařízení pomocí USB kabelu, či emulátoru. Při vývoji aplikací cílených pouze na OS Android je často využíváno programu AVD (*Android Virtual Device*) manager. Tento program je dostupný i ve Visual Studiu. S jeho pomocí lze vytvořit virtuální zařízení s nastavením velikosti paměti RAM (*Random Access Memory*), mapování SD karty a další. U některých typů zařízení lze zvolit rodinu procesoru (x86 či ARM(*Advanced RISC Machine*)). Visual Studio uvádí, že emulátory založené na ARM jsou zhruba desetkrát pomalejší. Vhodnější variantou je použití nástroje Visual Studio Emulator for Android dostupného z [10]. Po jeho spuštění je možné stahovat emulátory podobné konkrétním zařízením a to jak chytrých telefonů, tak tabletů. Zařízení jsou navíc řazena dle verze API. Po spuštění emulátoru je k dispozici stejný panel nástrojů, jaký byl popsán pro emulátory mobilních zařízení s Windows 10 a je zobrazen na Obr. 24.



Obr. 24 Rozšířený panel nástrojů emulátorů

Závěr

V diplomové práci byl popsán princip vývoje multiplatformních aplikací. Speciální pozornost byla věnována prostředí Xamarin.Forms. Byl popsán jeho vývoj a následně i jeho možnosti. Velká část práce byla věnována dostupným visuálním elementům. Dále byly rozebrány mnohé jmenné prostory Xamarin.Forms včetně vysvětlení podpory platformě specifického API. Práce definovala oblasti využití dvou programovacích jazyků spjatých s platformou Xamarin.Forms a popsala jejich vzájemnou spolupráci. Stručně byly rozebrány i jiné varianty multiplatformních prostředí a bylo provedeno jejich srovnání.

V druhé části práce byla implementována komplexní aplikace vyvinutá pomocí Xamarin.Forms. Tato aplikace slouží k ovládání jednotek PT41. V první kapitole druhé části byl objasněn princip a funkce jednotek PT41. Byly uvedeny dostupné typy jednotek a možnosti jejich konfigurace. Aplikace PT41 vyžadovala použití dalších knihoven, které nejsou součástí Xamarin.Forms API a byly přidány pomocí NuGet balíčků. Všechny z nich byly popsány. Knihovny sloužily pro podporu návrhového vzoru MVVM, zobrazování dialogů, podporu databází typu SQLite a síťové komunikace. Velká pozornost byla věnována tvorbě vlastních grafických elementů pomocí knihovny Skia.Sharp. Aplikace obsahuje několik takto vytvořených elementů. Bylo popsáno, jaké výhody a nevýhody se pojí s vytvářením visuálních elementů pomocí této knihovny oproti přístupu s použitím vlastních rendererů či již existujících elementů. Dále byly rozebrány vytvořené jmenné prostory obsahující vlastní logiku aplikace. Rozebrán byl i komunikační protokol spojený s jednotkami PT41. Na závěr byl pomocí obrázků ukázán visuální vzhled vytvořených stránek a byla popsána funkcionalita view-modelů, které jsou s nimi svázány. Popsán byl i způsob vývoje vícejazyčných aplikací a persistentního ukládání proměnných. V poslední podkapitole byly porovnány dostupné nástroje pro vývoj universálních aplikací a aplikací Xamarin.Forms a byly rozebrány nástroje pro ladění s využitím fyzických i virtuálních zařízení.

Aplikace je funkční na zařízeních s OS Android a na zařízeních s OS Windows 10 a to jak desktopových, tak mobilních. Bylo dosaženo stavu, kdy platformě specifické projekty jsou velmi malé a drtivá většina kódu je sdílena, což je základní výhodou multiplatformních aplikací.

Literatura

[1] A guide to the 10 best cross platform mobile development tools. *Thinkapps* [online]. San Francisco, 2016 [cit. 2016-12-11]. Dostupné z: http://thinkapps.com/blog/development/develop-for-ios-v-android-cross-platform-tools/

[2] ACR User Dialogs for Xamarin and Windows. *Github* [online]. GitHub, 2017 [cit. 2017-04-28]. Dostupné z: https://github.com/aritchie/userdialogs

[3] ALBAHARI, Joseph a Ben ALBAHARI. *C# 6.0 in a nutshell: the definitive reference*. Sixth edition. Sebastopol, CA: O'Reilly Media, 2015. In a nutshell (O'Reilly & Associates). ISBN 1491927062.

[4] ALLEN, Grant. *Android 4: průvodce programováním mobilních aplikací*. Brno: Computer Press, 2013. ISBN 978-80-251-3782-6.

[5] *Best free stock photos in one place* [online]. [cit. 2016-12-11]. Dostupné z: https://www.pexels.com/

[6] BROWN, Michael. *MVVM unleashed*. Indianapolis, Ind: Sams, 2012. ISBN 9780672334382.

[7] Browse Files. *Developer.xamarin.com* [online]. 2016 [cit. 2016-12-11]. Dostupné z: https://developer.xamarin.com/recipes/android/data/files/browse_files

[8] Cross-Platform 2D Graphics with SkiaSharp. *Xamarin blog* [online]. Xamarin, 2017 [cit. 2017-04-28]. Dostupné z: https://blog.xamarin.com/cross-platform-2d-graphics-with-skiasharp/

[9] *Elektrobock* [online]. Kuřim: Elektrobock, 2017 [cit. 2017-04-28]. Dostupné z: https://www.elektrobock.cz/

[10] Emulátor Androidu pro Visual Studio. *Www.visual studio* [online]. Microsoft, 2017 [cit. 2017-05-12]. Dostupné z: https://www.visualstudio.com/cs/vs/msft-android-emulator/

[11] Getting started with Prism. *Prismlibrary.readthedocs.io* [online]. 2017 [cit. 2017-04-28]. Dostupné z: http://prismlibrary.readthedocs.io/en/latest/Xamarin-Forms/1-Getting-Started/

[12] *Gorilla Player* [online]. UXDrivers [cit. 2017-05-12]. Dostupné z: http://gorillaplayer.com/

[13] PETZOLD, Charles. *CHARLES PETZOLD Cross-platform C# programming for iOS, Android, and Windows* [online]. Redmond, Washington 98052-6399: Microsoft Press, 2016 [cit. 2016-12-11]. ISBN 978-1-5093-0297-0. Dostupné z: https://download.microsoft.com/DOWNLOAD/7/8/8/788971A6-C4BB-43CA-91DC-557B8BE72928/MICROSOFT_PRESS_EBOOK_CREATINGMOBILEAPPSWITHXAMARINFORMS_PDF.PDF

[14] Prism Template Pack. *Marketplace.visualstudio* [online]. Brian Lagunas, 2017 [cit. 2017-04-28]. Dostupné z: https://marketplace.visualstudio.com/items?itemName=BrianLagunas.PrismTemplatePack

[15] Převodník RS232 na Ethernet/WiFi. *Elektrobock* [online]. Kuřim: Elektrobock, 2017 [cit. 2017-04-28]. Dostupné z: https://www.elektrobock.cz/prevodnik-rs232-na-ethernet-wifi/p270

[16] ResXManager. *Marketplace* [online]. Microsoft, 2017 [cit. 2017-04-28]. Dostupné z: https://marketplace.visualstudio.com/items?itemName=TomEnglert.ResXManager

[17] ScnViewGestures. *GitHub* [online]. GitHub, 2017 [cit. 2017-04-28]. Dostupné z: https://github.com/ScienceSoft-Inc/ViewGestures

[18] SerialArduino. *Github* [online]. GitHub, 2017 [cit. 2017-04-28]. Dostupné z: https://github.com/Microsoft/Windows-universal-samples/tree/master/Samples/SerialArduino

[19] Sockets-for-pcl. *Github* [online]. GitHub, 2017 [cit. 2017-04-28]. Dostupné z: https://github.com/rdavisau/sockets-for-pcl

[20] Sqlite-net. *Github* [online]. GitHub, 2017 [cit. 2017-04-28]. Dostupné z: https://github.com/praeclarum/sqlite-net

[21] TABOR, Bob a Steven NIKOLIC. *Windows 10 Development for Absolute Beginners* [online]. [cit. 2016-12-11]. Dostupné z: https://www.google.cz/url?sa=t&rct=j&q=&esrc=s&source=web&cd=5&ved=0ahUKEwi1gsmWgO3QAhWE5xoKHZuUABAQFghFMAQ&url=https%3A%2F%2Fwindowsdeveloper.azureedge.net%2Fpdfs%2FUniversal%2520Windows%2520Platform%2520for%2520Absolute%2520Beginners.pdf&usg=AFQjCNFQOnbwYLQRUTvTO_dIARtukI0Fjw&sig2=xQu3m8q-gyueSv1E8vPwng&cad=rja

[22] Ten of the Best Cross-Platform Mobile Development Tools for Enterprises. *Businessofapps* [online]. 2014 [cit. 2016-12-11]. Dostupné z: http://www.businessofapps.com/ten-best-cross-platform-development-mobile-enterprises/

[23] Windows SDK and emulator archive. *Any Developer*. *Any App*. *Any Platform* [online]. Microsofr [cit. 2017-05-12]. Dostupné z: https://developer.microsoft.com/en-us/windows/downloads/sdk-archive

[24] *Xamarin.Forms* [online]. 2016 [cit. 2016-12-11]. Dostupné z: https://www.xamarin.com/forms

[25] Xamarin.Forms.Easing Class. *Developer.xamarin.com* [online]. 2016 [cit. 2016-12-11]. Dostupné z: https://developer.xamarin.com/api/type/Xamarin.Forms.Easing

Použité zkratky

2D	Two-Dimensional
3D	Three-Dimensional
API	Application Programming Interface
ARM	Advanced RISC Machine
AVD	Android Virtual Device
CSS	Cascading Style Sheets
DHCP	Dynamic Host Configuration Protocol
DIP	Device Independent Pixel
DIU	Device Independent Unit
DPI	Density per Inch
DPS	Density-Independent Pixel
GUI	Graphical User Interface
HTML	HyperText Markup Language
IDE	Integrated Development Environment
IoC	Inversion of Control
IoT	Internet of Things
IP	Internet Protocol
IT	Information Technology
LED	Light-Emitting Diode
LINQ	Language Integrated Query
MAC	Apple Macintosh Computer
MVC	Model-view-controller
MVVM	Model View View-Model
NFC	Near Field Communication
OS	Operation System
PCL	Portable Class Library
PCL	Portable Class Library

PI	Proportional Integral
PPI	Pixels per Inch
RAM	Random Access Memory
SAP	Shared Asset Project
SAP	Shared Asset Project
SD	Secure Digital
SDK	Software Development Kit
SQL	Structured Query Language
TCP	Transmission Control Protocol
TLS	Transport Layer Security
USB	Universal Serial Bus
UWP	Universal Windows Platform
WPF	Windows Presentation Fundation
XAML	Extensible Application Markup Language
XML	eXtensible Markup Language

Seznam příloh na CD

1. Zdrojový kód aplikace PT41.